1. Describe and analyze an algorithm to compute a donation schedule, describing how much money each voter should send to each candidate on each day, that guarantees that every candidate gets enough money to win their election. The schedule must obey both Federal laws and individual voters' budget constraints. If no such schedule exists, your algorithm should report that fact.

   Assume there are n candidates, p party members, and d days until the election. The input to your algorithm is a pair of arrays $Win[1..n]$ and $Limit[1..p, 1..n]$, where $Win[i]$ is the amount of money candidate i needs to win, and $Limit[i, j]$ is the total amount party member i is willing to donate to candidate j.

   Your algorithm should return an array $Donate[1..p, 1..n, 1..d]$, where $Donate[i, j, k]$ is the amount of money party member i should donate to candidate j on day k.

Solution (triple selection, 8/10): We solve this as a triple-selection problem for days, voters, and candidates. Specifically, we construct a graph G with the following vertices and edges:

- G has a source vertex s, a vertex d_i for each day between today and election day, a vertex p_j for each party member, a vertex c_k for each candidate, and a target vertex t.
- For each day i, there is an edge $s \to d_i$ with infinite capacity.
- For each day i and party member j, there is an edge $d_i \to p_j$ with capacity 100.
- For each party member j and candidate k, there is an edge $p_j \to c_k$ with capacity $Limit[j, k]$. (To simplify the solution, I will assume this number is an integer.)
- Finally, for each candidate k, there is an edge $c_k \to t$ with capacity $Win[k]$. (To simplify the solution, I will also assume this number is an integer.)

Altogether this graph has $O(d + p + n)$ vertices and $O(p(d + n))$ edges.

To keep the rest of the solution clean, I'll permute the indices of the output array $Donate[1..d, 1..p, 1..n]$ to be consistent with the part order in G.

Once we construct the graph G, the rest of the algorithm proceeds as follows: First we compute a maximum (s, t)-flow f. If f does not saturate all edges into t, we halt and report failure. Otherwise, we decompose f into paths. Finally, for each triple of indices i, j, k, we set $Donate[i, j, k]$ to the weight of the path $s \to d_i \to p_j \to c_k \to t$ in the decomposition of f.

Say that a donation schedule is *successful* if all laws and budget constraints are satisfied and every candidate receives *exactly* their donation target. Our algorithm rests on the following claim: **There is a successful donation schedule if and only if our maximum flow f saturates every edge into t.** As usual, we prove this if-and-only-if claim in two parts:

- $\Leftarrow$ Suppose our maximum flow f saturates every edge into t.
  - Because f is feasible, we have $f(d_i \to p_j) \le c(d_i \to p_j) = 100$ for every day i and party member j, which implies $Donate[i, j, k] \le 100$ for all i, j, k. No broken laws!

- Because f is feasible, we have $0 \le f(p_j \to c_k) \le c(p_j \to c_k) = \mathit{Limit}[j,k]$ for each party member j and candidate k. Nobody goes over their donation limits!
- Finally, because f saturates every edge into t, we have $\sum_{i,j} \mathit{Donate}[i,j,k] = f(c_k \to t) = c(c_k \to t) = \mathit{Win}[k]$ for every candidate k. Everybody gets elected!

$\Leftarrow$ On the other hand, suppose the output array *Donate* describes a successful donation schedule. Define a flow f' as a weighted sum of paths as follows:

$$f' = \sum_{i,j,k} \mathit{Donate}[i,j,k](s \to d_i \to p_j \to c_k \to t)$$

We can verify that f' is a feasible flow that saturates every edge into f as follows:

- For each day i and party member j, we have $\sum_k \mathit{Donate}[i,j,k] \le 100$, and therefore $f'(d_i \to p_j) = \sum_k \mathit{Donate}[i,j,k] \le 100c(d_i \to p_j)$.
- For each party member j and candidate k, we have $\sum_i \mathit{Donate}[i,j,k] \le \mathit{Limit}[j,k]$ and therefore $f'(p_j \to c_k) = \sum_i \mathit{Donate}[i,j,k] \le \mathit{Limit}[j,k] = c(p_j \to c_k)$.
- For each candidate k, we have $\sum_{i,j} \mathit{Donate}[i,j,k] = \mathit{Win}[k]$ and therefore $f'(c_k \to t) = \sum_{i,j} \mathit{Donate}[i,j,k] = \mathit{Win}[k] = c(c_k \to t$.

Because f' saturates every edge into t, it must be a *maximum* flow. It follows that *every* maximum flow in G saturates every edge into t.

We can compute the maximum flow f in $O(VE)$ time using Orlin's algorithm,[a] decompose f into paths in $O(VE)$ time, and then transcribe the paths into the output array in $O(dpn)$ time. The overall running time of the algorithm is $O(VE + dpn) = \boldsymbol{O((d+p+n)p(d+n))}$. ■

[a] The maximum flow value is at most $100dp$, so Ford-Fulterson would compute the maximum flow in $O(dpE)$ time, leading to an overall running time of $O(dp^2(d+n))$.

Solution (pair selection, 10/10): We construct a flow network G with the following vertices and edges:

- G has a source vertex s, one vertex p_i for each party member, one vertex c_j for each candidate, and a target vertex t.
- For each party member i, there is an edge $s \to p_i$ with capacity $100d$, where d is the number of days until the election.
- For each party member i and candidate j, there is an edge $p_i \to c_j$ with capacity $\mathit{Limit}[i,j]$.
- Finally, for each candidate j, there is an edge $c_j \to t$ with capacity $\mathit{Win}[j]$.

Altogether this graph has $O(p+n)$ vertices and $O(pn)$ edges.

Once we construct G, the rest of the algorithm proceeds as follows: First we compute a maximum (s,t)-flow f. If f does not saturate all edges into t, we halt and report failure. Otherwise, we decompose f into paths and set $\mathit{Total}[i,j]$ to the

weight of the path $s \to d_i \to p_j \to c_k \to t$ in the decomposition; $Total[i, j]$ is the total amount of money that party member i will donate to candidate j. Finally, we distribute all donations across d days as follows:

```
DISTRIBUTE(Total[1..p, 1..n], d):
  for i ← 1 to p
      for j ← 1 to n
          for k ← 1 to d
              Donate[i, j, k] ← 0
  for i ← 1 to p
      j ← 1
      k ← 1
      while j ≤ n and k ≤ d
          Δ = min{Total[i, j], 100 − Donate[i, j, k]}
          Total[i, j] ← Total[i, j] − Δ
          Donate[i, j, k] ← Donate[i, j, k] + Δ
          if Total[i, j] = 0
              j ← j + 1
          else
              k ← k + 1
  return Donate
```

Say that a donation schedule is *successful* if all laws and budget constraints are satisfied and every candidate receives *exactly* their donation target. Our algorithm rests on the following claim: **There is a successful donation schedule if and only if our maximum flow f saturates every edge into t.** As usual, we prove this if-and-only-if claim in two parts:

- $\Leftarrow$ Suppose f saturates every edge into t. Definition-chasing implies the following inequalities:
  - $\sum_j Total[i, j] < 100d$ for all i – Nobody donates more than $100d$ dollars. The DISTRIBUTE algorithm splits each party members' donations across $\lceil \sum_j Total[i, j]/100 \rceil \le d$ days, with at most 100 dollars each day. No laws broken!
  - $Total[i, j] < Limit[i, j]$ for all i and j – Nobody goes over budget!
  - $\sum_i Total[i, j] = Win[j]$ for all j – Everyone gets elected!
- $\Leftarrow$ On the other hand, suppose the output array *Donate* describes a successful donation schedule. For all indices i and j, let $Total[i, j] = \sum_{k=1}^{d} Donate[i, j, k]$. Define a flow f as a weighted sum of paths as follows:

$$f' = \sum_{i,j} Total[i, j](s \to p_i \to c_j \to t)$$

  Routine definition-chasing implies that f' is a feasible flow that saturates every edge into t. It follows that f' is a maximum flow, and *every* maximum flow in G saturates every edge into t.

We can compute the maximum flow in G in $O(VE)$ time using Orlin's algorithm,[a] decompose it into paths in $O(VE)$ time, and then transcribe the paths into the output array in $O(dpn)$ time. The overall running time of the algorithm is $O(VE + dpn) = \mathbf{O((d + p + n)pn)}$. ■

[a]The maximum flow value is at most $100dp$, so Ford-Fulkerson would compute the maximum flow in $O(dpE) = O(dp^2n)$ time, leading to an overall running time of $O(dp^2n)$.

Rubric: 10 points: standard reduction rubric (3 for graph + 3 for explicit connection to given problem + 2 for other algorithmic details + 2 for time analysis). For purposes of comparing running times, assume $p = n = d$. Full credit for $O(n^3)$; max 8 points for $O(n^4)$; max 6 points for $O(n^5)$.

2. A ***k-orientation*** of an undirected graph G is an assignment of directions to the edges of G so that every vertex of G has at most k incoming edges. Describe and analyze an algorithm that determines the smallest value of k such that G has a k-orientation, given the undirected graph G as input.

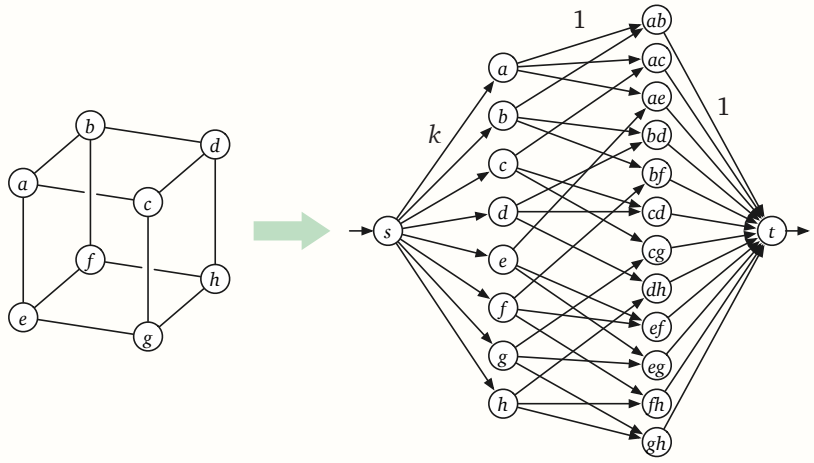
Solution: Our algorithm performs a binary search for the smallest k such that G has a k-orientation; for each value of k we consider, we intuitively look for an ***assignment*** of at most k incoming edges to each vertex. More concretely, we solve the decision problem as a generalized matching or pair-selection problem, where the two resource sets are the vertices and edges of G.

Fix an arbitrary value of k. To decide whether G has a k-orientation, we construct a flow network $H = (V', E')$ as follows:

- $V' = V \cup E \cup \{s, t\}$. Except for the source s and target t, the vertices of H correspond to the vertices *and edges* of G. Clearly $|V'| = 2 + |V| + |E| = O(E)$.
- E' contains three types of edges:
  - An edge $s \to v$, for each vertex $v \in V$.
  - An edge $v \to e$, for each edge $e \in E$ and each endpoint v of e.
  - An edge $e \to t$, for each edge $e \in E$.

  Altogether we have $|E'| = |V| + 2|E| + |E| = O(E)$.
- Each edge $s \to v$ has capacity k; all other edges have capacity 1.

The following figure shows the resulting flow network for the cube graph:



Our construction guarantees a correspondence between k-orientations of G and integer (s, t)-flows in H that saturate every edge into t; specifically, each flow path from s to t in H corresponds to a choice of direction for one edge in G.

- For any k-orientation of G, we can construct an integer flow f in H as follows. For each directed edge $u \to v$ in the orientation of G, we send one unit of flow through h along the path $s \to v \to uv \to t$; the flow f is the sum of these E paths. Because each vertex of G has at most k incoming edges, we have $f(s \to v) \le k$ for

every vertex v. Because each edge uv is either oriented into v or not, we have $f(v{\to}uv) \le 1$. Finally, because each edge of G has exactly one orientation, we have $f(e{\to}t) = 1$ for every edge E. We conclude that f is a feasible flow in H that saturates every edge into t.

- On the other hand, let f be any integer flow in H that saturates every edge into t. We can decompose f into E paths of the form $s{\to}v{\to}uv{\to}t$, each carrying one unit of flow. For each such path, assign edge uv the direction $u{\to}v$. Because $f(e{\to}t) = 1$ for every edge e in G, every edge e in G is assigned a unique direction. Because $f(s{\to}v) \le k$ for every vertex v of G, at most k edges in G are directed into v. So we have constructed a k-orientation of G.

Thus, to solve the decision problem for any fixed k, we construct the flow network H as described above, compute a maximum (s,t)-flow f^* in H, and then report success if and only if $|f^*| = E$. If we use Orlin's algorithm to compute the maximum flow, the decision algorithm runs in $O(V'E') = O(E^2)$ time.[a][b]

Finally, to solve the optimization problem, we perform a binary search over all possible values of k. Every graph has a V-orientation, and no graph has a (-1)-orientation, so we can limit our search to the range $0 \le k \le V-1$. It follows that our binary search requires $O(\log V)$ iterations, and thus our entire algorithm runs in $\boldsymbol{O(E^2 \log V)}$ ***time***. ■

[a] If we used Ford-Fulkerson here, our decision algorithm would run in $O(E^2 k)$ time, and so the resulting optimization algorithm would run in $O(E^2 V \log V)$ time.

[b] For an even faster decision algorithm, see: Yuichi Asahiro, Eiji Miyano, Hirotaka Ono, and Kouhei Zenmyo. Graph orientation algorithms to minimize the maximum outdegree. *Int. J. Found. Comput. Sci* 18(2):197–215, 2007.

Rubric: 10 points; standard reduction rubric. The proof of correctness (in gray) is not required for full credit. +5 for a correct $O(E^2)$-time algorithm. Even this is not the fastest algorithm for this problem.

Solution (extra credit: parametric flow): Instead of performing a binary search over all possible values of k, computing a maximum flow from scratch at each iteration, we consider all k from 1 up to the maximum, updating a maximum flow at each iteration.

We essentially the same flow network $H = (V', E')$ as in the previous solution:

- $V' = V \cup E \cup \{s,t\}$. Except for the source s and target t, the vertices of H correspond to the vertices *and edges* of G. Clearly $|V'| = 2 + |V| + |E| = O(E)$.
- E' contains three types of edges:
  - An edge $s{\to}v$, for each vertex $v \in V$.
  - An edge $v{\to}e$, for each edge $e \in E$ and each endpoint v of e.
  - An edge $e{\to}t$, for each edge $e \in E$.

  Altogether we have $|E'| = |V| + 2|E| + |E| = O(E)$.
- Initially *every* edge in this network has capacity 1.

We now proceed in several rounds. In the kth round, we set the capacity of all edges into t to k and compute a maximum flow, starting with the maximum flow from the previous round. If the maximum flow saturates all edges into t, that flow corresponds to a k-orientation of G, so we can stop and return k. Otherwise, G does not have a k-orientation, so we proceed to the $(k+1)$th round.

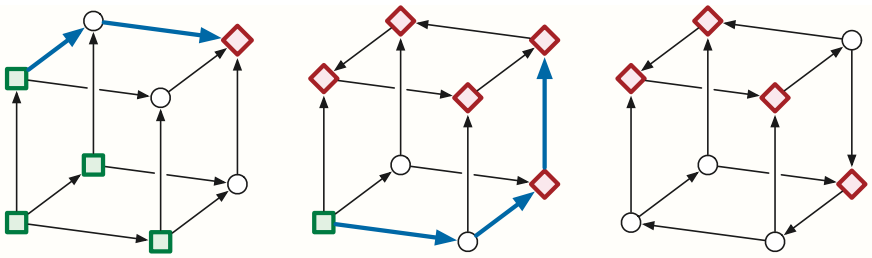
```
MinOrientation(V, E):
  Build the graph H as described above
  f ← 0        ⟨⟨flow corresponding to partial orientation⟩⟩
  for k ← 1 to V − 1
      for every vertex v ∈ v
          c(s→v) ← k
      while H_f contains a path from s to t
          P ← any path in H_f from s to t
          f = f + P        ⟨⟨push 1 unit of flow along P⟩⟩
      if f(e→t) = 1 for every edge e ∈ E
          return k
```

Finding each path P takes $O(V' + E') = O(E)$ time. Each time we push along a path P from s to t, we saturate one of the edges into t. Thus, the total number of pushes in the entire algorithm is at most E, the number of edges into t. (Equivalently: Every push increases the value of the flow by 1, and the maximum value of the flow is at most E, because the total capacity of all edges into t is E.) So the entire algorithm runs in $\boldsymbol{O(E^2)}$ ***time***. ■

Solution (extra credit; greedy improvement): The following algorithm is due to Venkat Venkateswaran (Minimizing maximum indegree. *Discrete Appl. Math.* 143(1-3): 374–378, 2004).

```
MinOrientation(G):
  arbitrarily orient the edges of G
  repeat forever
      k ← max{indeg(v) | v ∈ V}
      Hi ← {v ∈ V | indeg(v) = k}
      Lo ← {v ∈ V | indeg(v) ≤ k − 2}
      if there is no directed path in G from Lo to Hi
          return k
      P ← any directed path in G from Lo to Hi
      reverse every edge of P
```

The integer k never increases between iterations of the main loop, so the set Lo never grows. At every iteration of the algorithm, the in-degree of one vertex in Lo increases, and the in-degree of one node in Hi decreases; otherwise, all in-degrees remain unchanged. As long as a vertex is in the set Lo, its in-degree can only increase. Thus, the number of iterations is at most the sum of the degrees (both in- and out-) of the vertices in the *initial* set Lo, which is trivially at most $2E$. Each iteration takes $O(E)$ time, so the overall algorithm runs in $\boldsymbol{O(E^2)}$ ***time***.



Orienting the cube. Green square vertices are in *Lo*; red diamond vertices are in *Hi*.

We can prove this algorithm is correct as follows. Let U be the subset of vertices that are *not* reachable from the final set Lo in the final directed graph G. (In particular, if $Lo = \emptyset$, then $U = V$.) Let E_U be the set of directed edges in G whose heads (and therefore tails) are in U. Because the algorithm halted, we have $Hi \subseteq U$; every vertex in U has in-degree $k-1$ or k, and at least one vertex in T has in-degree k. Thus, $(k-1) \cdot |U| < |E_U| \leq k \cdot |U|$, which implies $k = \lceil |E_U|/|U| \rceil$. We conclude that in *every* orientation of G, some vertex in U has in-degree at least k. (In particular, some vertex in U has incoming edges from at least k other vertices in U.) ■

3. Let $G = (L \sqcup R, E)$ be a bipartite graph, whose left vertices L are indexed $\ell_1, \ell_2, \ldots, \ell_n$ and whose right vertices are indexed $r_1, r_2, \ldots, r_n$. A matching M in G is ***non-crossing*** if, for every pair of edges $\ell_i r_j$ and $\ell_{i'} r_{j'}$ in M, we have $i < i'$ if and only if $j < j'$. If we place the vertices of G in index order along two vertical lines and draw the edges of G as straight line segments, a matching is non-crossing if its edges do not intersect.

   Describe and analyze an algorithm to find the smallest number of disjoint non-crossing matchings $M_1, M_2, \ldots, M_k$ such that each edge in G lies in exactly one matching M_i.

   [Hint: How would you compute the largest non-crossing matching in G?]

Solution (disjoint-path cover): Given the input graph $G = (L \sqcup R, E)$, we construct a new directed acyclic graph $G' = (V', E')$ as follows:

- $V' = E$; that is, the vertices of G' correspond to the edges of G.
- $E' = \{\ell_i r_j \to \ell_k r_l \mid i < k \text{ and } j < l\}$. That is, two edges in G are connected by an edfge in G' if and only if they can appear in the correct order in the same non-crossing matching.

Every directed path in G' corresponds to a non-crossing matching in G and vice versa. Thus, the smallest set of disjoint non-crossing matchings that cover every edge of G corresponds to the smallest set of disjoint paths that cover every vertex of G'. We can compute the smallest disjoint-path cover of G' in $O(V'E') = \boldsymbol{O(E^3)}$ ***time***, using the algorithm described in the textbook.

G' is the dependency dag of the following recurrence for computing the largest non-crossing matching in G. Let $NXM(i, j)$ denote the size of the largest non-crossing matching containing the edge $\ell_i r_j$.

$$NXM(i,j) = 1 + \max\left\{NXM(k,l) \;\middle|\; i < k \text{ and } j < l \text{ and } \ell_k r_l \in E\right\}$$

(Here I implicitly assume $\max \varnothing = 0$.) The *largest* non-crossing matching corresponds to the *longest* path in the dependency dag G'. ■

Rubric: 10 points: standard reduction rubric. This is not the only correct algorithm with this running time. This is not the fastest algorithm for this problem. Max 13 points for $O(E^2)$ time; max 16 points for $O(E \log E)$ time; max 20 points for $O(E \log\log E)$ time; partial credit at the grader's discretion.

Solution (extra credit: chain–anti-chain duality): This solution will use some classical results about partial orders. Since not everyone has seen these objects before, I need to start with some definitions.

A *(weak) partial order* over a set X is a binary relation $\preceq$ satifiying three properties:

- Reflexive: $x \preceq x$ for all $x \in X$.
- Anti-symmetric: $x \preceq y$ and $y \preceq x$ implies $x = y$.
- Transitive: $x \preceq y$ and $y \preceq z$ implies $x \preceq z$.

Two elements $x, y \in X$ are *comparable* with respect to $\preceq$ if $x \preceq y$ or $y \preceq x$, and *incomparable* otherwise. A *chain* is a subset $C \subseteq X$ that is totally ordered by $\preceq$; that is, every pair of elements in C is comparable. An *anti-chain* is a subset $A \subseteq X$ in which every pair of elements is incomparable.

In graph-theoretic terms, the reachability relation defines a partial order over any directed acyclic graph. (A chain in G is a subset of vertices that lie on a common directed path in G. An anti-chain in G is a subset A of vertices such that any directed path in G touches at most one vertex in A.) Conversely, every partially ordered set $(X, \preceq)$ defines a directed acyclic graph, whose vertices are X and whose edges $x \to y$ are ordered pairs of distinct vertices $x \neq y$ such that $x \preceq y$.

Two fundamental theorems relate chains and anti-chains in arbitrary partial orders.

- **Dilworth's theorem:** The minimum number of disjoint chains that cover X is equal to the size of the largest anti-chain in X.
- **Mirsky's theorem:** The minimum number of disjoint anti-chains that cover X is equal to the size of the largest chain in X.

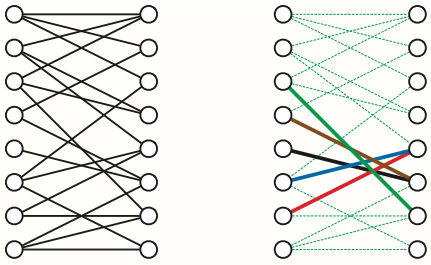
Dilworth's theorem can actually be proved using the maxflow-mincut theorem, but Mirsky's theorem has a much more straightforward proof.[a] The set X^- of all $\preceq$-minimal elements of X is an anti-chain. The longest chain in X contains exactly one element in X^-. Release the induction hounds on $X \setminus X^-$.

Let $G = (L \sqcup R, E)$ be the input bipartite graph. We define two partial orders $\preceq$ and $\not\prec$ over the edges of G as follows:

$$\ell_i r_j \preceq \ell_k r_l \iff (i = k \text{ and } j = l) \text{ or } (i < k \text{ and } j < l)$$
$$\ell_i r_j \not\prec \ell_k r_l \iff i \leq k \text{ and } j \geq l$$

Chains in $\preceq$ and anti-chains in $\not\prec$ are non-crossing matchings. Anti-chains in $\preceq$ and chains in $\not\prec$ are subgraphs of G in which *every* pair of edges intersects, either by crossing or by sharing an endpoint. Let's call these subgraphs *tangled pairings*.

Both Dilworth's theorem (applied to $\preceq$) and Mirsky's theorem (applied to $\not\prec$) imply that the minimum number of non-crossing matchings that cover G is equal to the size of the largest tangled pairing. The largest tangled pairing in G corresponds to the longest path in the dag $(E, \not\prec)$, which has E vertices and $O(E^2)$ edges. We can construct this dag by brute force and then compute its longest path in $\boldsymbol{O(E^2)}$ ***time*** via dynamic programming or depth-first search.



The largest tangled pairing in a bipartite graph.
Compare with the non-crossing matchings in the homework handout.

■

[a]Dilworth's theorem, which he published in 1950, is an immediate consequence of a theorem published in 1960 by Tibor Gallai and James Milgram: The vertices of any directed graph G can be partitioned into $\alpha(G)$ vertex-disjoint paths, where $\alpha(G)$ is the size of the largest independent set in G. However, Gallai and Milgram actually proved their theorem in the late 1940s, so some people attribute Dilworth's theorem to them. On the other hand, James Dilworth *also* knew his eponymous theorem years before he published it, so there, ha. All of these people, along with Fulkerson and others, knew Mirsky's theorem as well, but considered it too trivial to publish. Reportedly, when Leon Mirsky published it in 1971, Dilworth uttered one of the two curse words he ever said in his life.

Solution (extra extra credit: faster dynamic programming): But we can compute the largest tangled pairing in G more quickly.

Draw the input graph G in the plane with vertices in L on the vertical line $x = 0$, vertices in R on the vertical line $x = 1$, and edges drawn as line segments. Extend the segments in E very slightly to the vertical lines $x = -\varepsilon$ and $x = \varepsilon$ for some sufficiently small $\varepsilon > 0$; call the resulting segments S. Sort the segments in S by the y-coordinates of their left endpoints, and let $R[1..E]$ denote the y-coordinates of the corresponding right endpoints. Building the array R takes $O(E \log E)$ time.

For any indices $i < j$, the ith and jth segments in S intersect if and only if $R[i] > R[j]$. On the other hand, these two segments intersect if and only if the original edges either cross of share an endpoint. Thus, every tangled pairing in G corresponds with a decreasing subsequence in R. In particular, the largest tangled pairing in G corresponds with the ***longest decreasing subsequence*** in R.

We can compute the length of the longest decreasing sequence in R in $\boldsymbol{O(E \log E)}$ ***time*** as follows. We scan through the array E in order, maintaining an array $LLE[0..E]$, where $LLE[j]$ is the *L*argest possible *L*ast *E*lement of a decreasing subsequence of length j of the elements scanned so far, or $-\infty$ if there is no such subsequence.

```
LDS(R[1..E]):
  for j ← 1 to E
      LLE[j] ← −∞                    ⟨⟨sentinel⟩⟩
  for i ← 1 to n
      ℓ ← min{j | LLE[j] ≥ R[E]}     ⟨⟨binary search⟩⟩
      LLE[ℓ] ← R[i]
  ℓ ← max{j | LLE[j] > −∞}           ⟨⟨binary search⟩⟩
  return ℓ
```

The two key observations are that the *LLE* array is always sorted in decreasing order, and that the array changes in exactly one position at each iteration of the main loop. Specifically, at the ith iteration, $R[i]$ replaces the leftmost value $LLE[\ell]$ smaller than $R[i]$. At each iteration, we are using binary search to find which entry $LLE[\ell]$ to update.

When the algorithm ends, the final *LLE* array (ignoring sentinel elements) it has the same length as the longest decreasing subsequence of R. (However, it is *not* itself a decreasing subsequence of R.)

For example, given the input array $[8,6,1,7,5,0,3,9,6,2,8,4]$, the *LLE* array would evolve as follows. (To save space, I'll omit the sentinel values at both ends.)

$$[] \rightsquigarrow [\mathbf{8}] \rightsquigarrow [8,\mathbf{6}] \rightsquigarrow [8,6,\mathbf{1}] \rightsquigarrow [8,\mathbf{7},1] \rightsquigarrow$$
$$[8,7,\mathbf{5}] \rightsquigarrow [8,7,5,\mathbf{0}] \rightsquigarrow [8,7,5,\mathbf{3}] \rightsquigarrow [\mathbf{9},7,5,3] \rightsquigarrow$$
$$[9,7,\mathbf{6},3] \rightsquigarrow [9,7,6,3,\mathbf{2}] \rightsquigarrow [9,\mathbf{8},6,3,2] \rightsquigarrow [9,8,6,\mathbf{4},2]$$

And indeed, the longest decreasing subsequence of $[\mathbf{8},6,1,\mathbf{7},\mathbf{5},0,\mathbf{3},9,6,\mathbf{2},8,4]$ has length 5. ■

Solution (extra extra extra credit: faster ordered dictionary): Oh, you thought we were *done?*

First let's replace the array *LLE* with an ordered dictionary that supports the operations INSERT, DELETE, and PRED(ecessor). Specifically, PRED(S,x) returns the largest element of S that is less than x, or $-\infty$ if there no such element.

```
LDS(R[1..E]):
  LLE ← new ordered dictionary
  ℓ ← 0
  for i ← 1 to E
      INSERT(LLE, R[i])
      if PRED(LLE, R[i]) = −∞
          ℓ ← ℓ + 1
      else
          DELETE(LLE, PRED(LLE, R[i]))
  return ℓ
```

If we implement the ordered dictionary with either a sorted array or a balanced binary search tree, each operation takes $O(\log E)$ time (possibly amortized), so the algorithm again runs in $O(E \log E) = O(E \log n)$ time.

But ordered dictionaries can be implemented more efficiently if the items being stored are small integers! Specifically, if the keys are integers between 1 and N, a *van Emde Boas tree* can support each INSERT, DELETE, and PRED operation in only $O(\log \log N)$ time. The design and analysis of van Emde Boas trees assumes that integers are stored as bit-strings using their standard binary representations, that the

integer N fits in a single machine word, and that we can perform standard arithmetic, bitwise boolean, and bit-shifting operations on machine words in constant time.

A van Emde Boas tree for a set S of integers between 1 and N has the following components. For simplicity, I will assume that $N = 2^w = 2^{2^k}$, so each integer in S is $2^k = w = \log_2 N$ bits long.

- The minimum element $\min S$ and maximum element $\max S$ are stored explicitly. Let $S' = S \setminus \{\min S, \max S\}$.
- We can split any integer $z \in S'$ into two $w/2$-bit strings $z^+ = \lfloor z/\sqrt{N} \rfloor$ and $z^- = z \bmod \sqrt{N}$, so that $z = z^+ \cdot \sqrt{N} + z^-$. We recursively store a van Emde Boas tree of the set of top halves $S^+ = \{z^+ \mid z \in S'\}$.
- For any $w/2$-bit integer x^+, we also recursively store a van Emde Boas tree of the set of bottom halves $S^-[x^+] = \{z^- \mid z \in S' \text{ and } z^+ = x^+\}$.

We can then implement $\textsc{Pred}(S, z)$ for any integer z between 1 and N as follows:

```
PRED(S, z):
  if z ≤ min S
      return −∞
  else if S_{z+} = ∅ or z^− ≤ min S^−_{z+}
      y^+ = PRED(S^+, z^+)
      y^− = max S^−[y^+]
      return y^+ · √N + y^−
  else
      y^− = PRED(S[z^+], z^−)
      return z^+ · √N + y^−
```

The running time of this algorithm satisfies the recurrence $T(N) = O(1) + T(\sqrt{N})$, or equivalently $T(2^{2^k}) = O(1) + T(2^{2^{k-1}})$, which implies $T(2^{2^k}) = O(k)$ and therefore $T(N) = O(\log \log N)$. Intuitively, this algorithm is performing a binary search *over the bits of* z for the longest prefix of z that agrees with a prefix of some element $y \in S$.

The INSERT and DELETE algorithms are similar: Split the input key z into the top half $z^+ = \lfloor z/\sqrt{N} \rfloor$ and bottom half $z^- = z \bmod \sqrt{N}$. After a constant amount of overhead, we recursively INSERT or *Delete* either in the set S^+ of top halves or in one subset $S^-[z^+]$ of bottom halves, but (because the min and max elements are stored separately), never into both.

You can find a more detailed description of van Emde Boas trees, including descriptions of the INSERT and DELETE algorithms, on Wikipedia.

The following algorithm sets up an array R that contains small integers. I'm assuming that the input graph is stored in a standard adjacency list, each vertex stores its index in either L or R, and each edge has pointers to its left and right endpoints. However, I am *not* assuming that either the vertex array(s) or the neighbor/edge lists are sorted in any particular order.

```
INITIALIZER(V, E):
  for each edge e in E
      i ← e.left.index
      j ← e.right.index
      e.L ← n · i − j + 1
      e.R ← n · j − i + 1
  sort E by e.L          ⟨⟨O(E log log n) time via vEB⟩⟩
  for k ← 1 to E
      R[k] ← E[k].R
  return R[1..E]
```

Sorting the edges by $e.L$ yields the same order as sorting the extended segments by their left y-coordinates in the first dynamic programming solution. Similarly, the $e.R$ values encode the order of the right endpoints of those segments.

For each edge e, we have $1 \le e.L \le n^2$, so we can sort E by $e.L$ in $O(E \log\log n)$ time using a van Emde Boas tree as a priority queue: INSERT all E edges using $e.L$ as the search key, and then repeatedly DELETE the edge with the largest search key (identified by PRED(∞)). Alternatively, we can sort E using base-n radix sort in only $O(E)$ time, but that's boring.

For each edge e, we also have $1 \le e.R \le n^2$. So if we implement the ordered dictionary in the main algorithm as a van Emde Boas tree, the main algorithm also runs in $\boldsymbol{O(E \log\log n)}$ ***time***. ■

4. **[just for practice, not for submission]** Suppose we are given a chessboard with certain squares removed, represented as a two-dimensional boolean array $A[1..n, 1..n]$. Describe and analyze efficient algorithms to place as many chess pieces of a given type onto the board as possible, so that no two pieces attack each other. A piece can be placed on the square in row i and column j if and only if $A[i, j] = \text{TRUE}$. Specifically:

   (a) Describe an algorithm to places as many *rooks* on the board as possible. A rook on square (i, j) attacks every square in the same row or column; that is, every square of the form (i, k) or (k, j).

   Solution (bipartite matching): We build a bipartite graph $G = (R \sqcup C, E)$ whose vertices correspond to the n rows and n columns of the original chessboard, and whose edges correspond to available squares. Every matching in G corresponds to a subset of the available squares with at most one in each row and at most one in each column, or in other words, a legal placement of rooks. In particular, the largest valid placement of rooks corresponds to the largest matching in G. We can compute this matching in $O(VE) = O(n^3)$ time. ■

   (b) Describe an algorithm to places as many *bishops* on the board as possible. A bishop on square (i, j) attacks every square on the same diagonal or back-diagonal; that is, every square of the form $(i + k, j + k)$ or $(i + k, j - k)$.

   Solution (bipartite matching): We build a bipartite graph $G = (D \sqcup B, E)$ whose vertices correspond to the $2n - 1$ diagonals and $2m - 1$ back-diagonals of the original chessboard, and whose edges correspond to available squares. Every matching in G corresponds to a subset of the available squares with at most one on each diagonal and at most one on each back-diagonal, or in other words, a legal placement of bishops. In particular, the largest valid placement of bishops corresponds to the largest matching in G. We can compute this matching in $O(VE) = O(n^3)$ time. ■

   *(c) Describe an algorithm to places as many *knights* on the board as possible. A knight on square (i, j) attacks the eight squares $(i \pm 1, j \pm 2)$ and $(i \pm 2, j \pm 1)$.

   Solution (bipartite independent set): We build a graph $G = (V, E)$ whose vertices correspond to avilable squares and and whose edges correspond to knight moves. G has at most n^2 vertices and (very crudely) at most $8n^2$ edges. Every independent set in G corresponds to a legal placement of knights. In particular, the largest valid placement of knights corresponds to the largest independent set in G.

   Color the available squares alternately black and white consistently with the original chessboard. A knight on a white square attacks only black squares, and vice versa; thus, G is actually a bipartite graph. Let W and B be the vertices of G corresponding to white and black squares, respectively.

   We can compute the largest independent set in G in $O(VE) = O(n^4)$ time as follows. Extend G to a flow network H by connecting a source s to W with unit-capacity edges, directing edges from W to B and giving them infinite capacity, and connecting B to a target vertex t with unit-capacity edges. Let f be the

maximum flow in H; let S be all vertices reachable from s in the residual graph H_f; let $T = V \setminus S$; and finally let $I = (W \cap S) \cup (B \cap T)$.

I proved in class that $(W \cup B) \setminus I$ is the smallest vertex cover of G. It follows by routine definition-chasing that I is the largest independent set in G. ■

★(d) Prove that placing as many *queens* on the board as possible is NP-hard. A queen attacks like either a rook or a bishop; that is, it attacks every square on the same row, column, diagonal, or back-diagonal.

Solution: See: Ian P. Gent, Christopher Jefferson, Peter Nightingale. Complexity of n-queens completion. *J. Artif. Intell. Res.* 59:815–848 (2017). https://eprints.whiterose.ac.uk/141513/

This paper actually proves that a *special case* of the maximum-independent-queens problem is NP-hard. Specifically, they consider inputs where the removed squares are those attacked by an independent set of queens on a complete $n \times n$ chessboard. I suspect that there is a simpler NP-hardness proof for arbitrary partial boards, probably from 3-dimensional matching. ■