Quiz 2

- Do not open this quiz booklet until you are directed to do so. Read all the instructions first.
- The quiz contains 6 problems, several with multiple parts. You have 2:15 hours for this midterm.
- This booklet contains a total of 16 pages, including this one and three sheets of scratch paper.
- Write your solutions in the space provided. If you run out of space, continue on a scratch page and make a notation.
- When we ask you to give an algorithm in this quiz, describe your algorithm in English or pseudocode, and provide a short argument for correctness and running time. You do not need to provide a diagram or example unless it helps make your explanation clearer.
- Do not waste time deriving facts that we have studied. Just cite results from class.
- Do not spend too much time on any one problem.

♣ **Do not remove any of these pages!**

♣ **Please write your name on every single page of this exam.**

Problem	Title	Points	Parts	Grade	Initials
0	Name	1	1		
1	True/False	8	4		
2	Short Answers	10	2		
3	Maximum Vertex Biclique	20	2		
4	Correcting Flows	20	2		
5	Sum of Minima	20	1		
6	Linear Program for Matchings	22	4		
Total		100			

NAME: _____

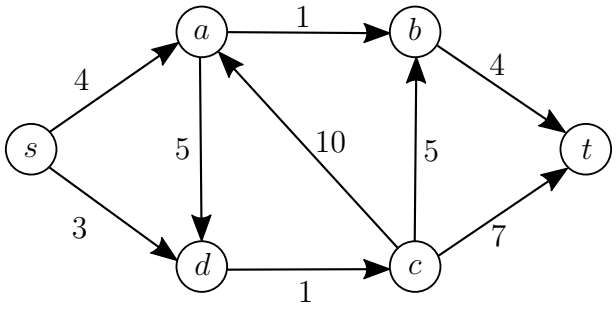
Circle your recitation:

F10	F11	F12	F1	F2	F3	F11	F12	F1
Katerina	Themis	Shankha	Maryam	Nishanth	Prashant	Manolis	Shalom	Ethan
R01	R02	R03	R04	R05	R06	R07	R08	R09

Quiz 2-1: [8 points] True/False Questions: Parts (a) – (d)

Mark each statement as either true or false. You have to **provide a very short explanation for each**.

(a) [2 points] In the following network, it is possible to send 3 units of flow from node s to node t.



Solution. **False.** Consider the (directed) S-T cut where $S = \{s, a, d\}, T = \{t, b, c\}$. The cut has size 2, and thus by the max-flow min-cut theorem which says that the max-flow from s to t is at most the size of the minimum S-T cut, we get that the max-flow is at most 2.

(b) [2 points] A problem $\Pi \in NP$ if and only if there exists a polynomial time algorithm V_Π such that: *For all* x, $\Pi(x) = yes$ *if and only if there exists a string* y *such that* $V_\Pi(x, y) = 1$.

Solution. **False.** The string y should also have size polynomial in the length of x.

(c) [2 points] If Π_1 can be reduced to Π_2 and Π_2 is NP-complete, then Π_1 is NP-complete.

Solution. **False.** Any problem in NP can be reduced to an NP-complete problem. So, Π_1 is in NP but not necessarily NP-hard and therefore not necessarily NP-complete.

(d) [2 points] Consider the 2-player game where each player has 3 possible strategies and has the following payoff matrix:

		Player A 1	Player A 2	Player A 3
Player B	a	0 0	+1 -1	-2 +2
	b	-1 +1	0 0	+1 -1
	c	+2 -2	-1 +1	0 0

The mixed strategy where

i. A chooses one of the strategies 1, 2, 3 with probability 1/3 each.

ii. B chooses one of the strategies a, b, c with probability 1/3 each.

is a Nash equilibrium of the described game.

Solution. **False.** With these mixed strategies, the payoff for each players is chosen uniformly at random from the entries of the 3 by 3 payoff matrix. Due to symmetry, the expected payoff for each player is 0. However, player B has incentive to change their strategy to playing (a) with probability 1. (Similarly, A has incentive to change their strategy to playing (1) with probability 1.) In this case, the expected payoff is: $\frac{1}{3} \cdot 0 + \frac{1}{3} \cdot (-1) + \frac{1}{3} \cdot 2 = \frac{1}{3} > 0$

Quiz 2-2: [12 points] Short Answers: Parts (a) – (b)

For each of the following questions you have to **provide an answer together with a short proof for your statement**.

(a) [6 points] A *planar graph* is a graph that can be drawn on the plane in such a way that its edges intersect only at their endpoints (that is, when drawn as straight line segments the edges do not cross each other). Given an undirected weighted planar graph $G = (V, E)$, can you solve All-Pairs-Shortest-Paths in G in $O(|V|^2 \log |V|)$ time?
Hint: You can use without proof that any planar graph has at least one vertex with degree at most 6.

Solution.

Let $n = |V|$ and $m = |E|$. It suffices to show that planar graphs are sparse i.e., $m = O(n)$, since we can then use Johnson's algorithm which runs in $O(nm + n^2 \log n) = O(n^2 \log n)$.

So, consider a vertex which has degree at most 6. If we remove this vertex, the graph still remains planar. So, the new graph also has another vertex of degree at most 6. If we continue removing the lowest degree vertex iteratively, we will end up removing all the edges of the graph in V iterations by removing at most 6 edges at a time. Thus, $m \leq 6n$. That means that the graph is sparse and concludes the proof.

(b) [6 points] Assume f is convex, and consider the following condition on $\vec{x}$, where $\vec{x} \in \mathbb{R}^n$:

$$\forall \vec{d} \in \mathbb{R}^n, \quad \textit{it holds that} \quad \langle \nabla f(\vec{x}), \vec{d} \rangle \geq 0 \quad (*)$$

We also remind you the convexity condition in high dimensions, i.e. $n > 1$. A function f is *convex* if and only if

$$\forall \vec{x}, \vec{y} \in \mathbb{R}^n, \quad f(\vec{y}) \geq f(\vec{x}) + \langle \nabla f(\vec{x}), \vec{y} - \vec{x} \rangle$$

Where we use $\langle \cdot, \cdot \rangle$ to denote the *dot product* in $\mathbb{R}^n$.
Explain with words why $(*)$ holds if and only if $\vec{x}$ is optimal, i.e. this is an *optimality condition*.

Solution. The condition $\langle \nabla f(\vec{x}), \vec{d} \rangle \geq 0$ is the same as saying that the derivative of f at $\vec{x}$ in the direction $\vec{d}$ is positive and therefore f increases if we make an infinitely small move in the direction of $\vec{d}$. But because this is true for all the possible directions we get that f increases if we make an infinitely small move in any direction and therefore $\vec{x}$ is a local minimum of f. But since f is convex it can only have one local minimum – namely, the global minimum – and therefore $\vec{x}$ is the global minimum of f. More formally, from the convexity of f, we get:

$$\forall \vec{y} \in \mathbb{R}^n, \quad f(\vec{y}) \geq f(\vec{x}) + \langle \nabla f(\vec{x}), \vec{y} - \vec{x} \rangle \geq^{(*)} f(\vec{x})$$

Thus, $\vec{x}$ is a global minimum.

Conversely, if $\vec{x}$ is a global minimum, then $\nabla f(\vec{x}) = \vec{0}$.

So,

$$\forall \vec{d} \in \mathbb{R}^n : \langle \nabla f(\vec{x}), \vec{d} \rangle = 0$$

Quiz 2-3: [20 points] Maximum Vertex Biclique: Parts (a) – (b)

We define the *biclique* $K_{a,b}$ with parameters a, b to be a bipartite graph with a vertices on one side, b vertices on the other side and edges between every vertex on one side to every vertex on the other side. More precisely

$$K_{a,b} = (A \cup B, \{\{u, v\} \mid u \in A, v \in B\}) \text{ with } |A| = a, \ |B| = b$$

We remind you that an *induced subgraph* G_S of a graph G is a graph that contains a subset S of the vertices of $G = (V, E)$ together with all edges whose endpoints are both in S. Formally

$$G_S = (S, \{\{u, v\} \mid \{u, v\} \in E \ \textit{and} \ u, v \in S\})$$

We also define an *empty graph* I_c with parameter c to be the graph

$$I_c = (S, \varnothing) \ \textit{with} \ |S| = c$$

An *independent set* of a graph $G = (V, E)$ is an induced subgraph of G that is an empty graph, i.e. an induced subgraph of G with c vertices and no edge between them.

Given an unweighted, undirected bipartite graph $G = (V = V_1 \cup V_2, E)$, our goal is to find the size of the induced subgraph of G that is a biclique with the maximum number of vertices possible. For convenience we let $n = |V|$ and $m = |E|$.

(a) [10 points] Let p be the size of the minimum vertex cover in G. Prove that the size of the maximum independent set in G is $n - p$.

Solution. Let $S \subseteq V$ be a minimum vertex cover of G. We know that $|S| = p$. Since S is a vertex cover, all edges have at least one endpoint in S. So, there are no edges with both endpoints in $T = V \setminus S$. That is, T is an independent set. So, there exists an independent set of size $|T| = |V \setminus S| = n - p$.

Now suppose that there exists another independent set I, such that $|I| > n - p$. Since I is an independent set, there are no vertices with both endpoints in I. Thus, $Q = V \setminus I$ is a vertex cover and $|Q| = n - |I| < p$. This is a contradiction since the minimum vertex cover has size p.

So, the size of the maximum independent set of G is exactly $n - p$.

A common mistake was to say that adding any vertex in the minimum vertex cover to $V \setminus S$ would break the independent set property. This is true but not enough to prove the claim.

(b) [10 points] Provide an algorithm that given G as input finds the size (number of vertices) of the induced subgraph of G with the maximum number of vertices that is a biclique. The running time of your algorithm should be $O(\sqrt{|V|}|E|)$.
Hint: If $H = (V_H, E_H)$ is a simple undirected graph, then we define the *complement* of H, denoted $\overline{H}$ to be the graph

$$\overline{H} = (V_H, \overline{E}_H = \{\{u, v\} \mid u, v \in V_H \text{ and } \{u, v\} \notin E_H\})$$

Define the bipartite version of the complement operation and use it along with part (a).

Solution. We define the *bipartite complement* of a bipartite graph $H = (L_H \cup R_H, E_H)$ as follows:

$$\tilde{H} = (V_H, \tilde{E}_H = \{\{u, v\} \mid u \in L_H, v \in R_H \text{ and } \{u, v\} \notin E_H\})$$

We observe that any biclique in G is an independent set in $\tilde{G}$. Also, any independent set in $\tilde{G}$ is a biclique in G. So, the maximum biclique in G is the maximum independent set in $\tilde{G}$.

Using part a, in order to find the size of the maximum independent set in $\tilde{G}$ it suffices to find the size of the minimum vertex cover in $\tilde{G}$, which is equal to the size of the maximum matching (Konig's theorem applies because $\tilde{G}$ is still bipartite).

To be explicit, given $G = (V, E)$ as input, the algorithm is:

- construct $\tilde{G}$
- run Hopcroft-Karp on $\tilde{G}$ to get a maximum matching; call the size of this matching p
- return $|V| - p$

The runtime of Hopcroft-Karp is $O(\sqrt{|V|}|\tilde{E}|)$ which is $O(|V|^{\frac{5}{2}})$, and this dominates the overall runtime, as finding $\tilde{G}$ takes $O(|V| + |\tilde{E}|)$ time.

Quiz 2-4: [20 points] Correcting Flows: Parts (a) – (b)

Let $G = (V, E, c)$ be a directed graph with *integer* capacities $c : E \to \mathbb{N}$. We also fix a source $s \in V$ and a sink $t \in V$. Suppose that you are given a valid flow for graph G , $f_G : E \to \mathbb{N}$, such that f_G pushes the maximum possible amount of flow from s to t. That is, f_G is a max flow in G.

Suddenly something happens to the flow network and the capacity $c(e_1)$ of one specific edge $e_1 \in E$ changes. We define the new capacity function $c' : E \to \mathbb{N}$ to be identical to c for every edge except for edge e_1 which changes to the new capacity $c'(e_1) \neq c(e_1)$.We also define $H = (V, E, c')$. Now we are not sure any more that f_G is a valid flow, and even if it is valid we are not sure that it is the optimal one. Our goal is to find as quickly as possible the new optimal flow f_H given G, H and f_G. For simplicity we set $n = |V|$, $m = |E|$ and $L = |c'(e_1) - c(e_1)|$. We denote by G_f the residual graph of G after sending flow f.

(a) [8 points] If we have the guarantee that $f_G(e_1) \leq c'(e_1)$, provide an algorithm that finds f_H given G, H, f_G. Prove that the running time of your algorithm is $O((n + m)L)$.

Solution. We distinguish the following cases:

- The edge e_1 was not saturated by the flow f_G in the initial graph:
  In this case, we know that f_G is a valid flow and also that s and t are disconnected in the residual graph which contains the edge e_1 with non-zero capacity. Changing the capacity of the edge is not going to affect the reachability of node t from s. So, the source s is also disconnected from the sink t in the residual graph H_{f_G}. Thus f_G is a maximum flow for graph H as well and we can output it immediately.
- The edge e_1 was saturated by the flow f_G in the initial graph:
  In this case, we know that there is no path from s to t in the residual graph G_{f_G}. However, the edge e_1 is not present in this graph (with the original orientation). So, if the capacity of this edge is increased, it will be present in H_{f_G} with capacity at most L. This addition of edge e_1 in the residual graph could potentially make t reachable from s again. However, the cut that separates s from t in G_{f_G}, will have capacity at most L in H_{f_G}. Therefore by the min-cut max-flow theorem we have that the maximum flow in H_{f_G} is at most L. Thus, the amount of extra flow that can be sent from s to t in H_{f_G} is at most L and since all the capacities are integers, the Ford-Fulkerson algorithm will do at most L augmentations. With that we conclude that the running time will be $O((n + m)L)$.

Now we assume that $c(e_1) \geq |f_G(e_1)| > c'(e_1)$.

(b) [12 points] Given f_G provide an algorithm that finds f_H (i.e the maximum flow in the new graph) in time $O((n+m)L)$.
Hint: As a first step find an algorithm that changes f_G to a valid flow for H. To do so find an algorithm that is given the residual graph G_{f_G} and pushes back at most L units of flow from t to s in a way that all this flow goes through the e_1 in the reverse direction.

Solution. We first construct the residual graph G_{f_G} which should have an edge $\bar{e_1}$ in the reverse direction of e_1 (and between the same nodes) of capacity $f_G(e_1)$ indicating that $f_G(e_1)$ units of flow currently go through edge e_1. Our goal is to find augmenting paths sending auxiliary flow f_{aux} from t to s such that the capacity of $\bar{e_1}$ in the new residual graph $G_{f_G+f_{aux}}$ is at most $c'(e_1)$. That means that $f_G + f_{aux}$ is a valid flow for graph H as well. Note that f_G is a flow from s to t, while f_{aux} is a flow in the reverse direction (i.e from t to s). Thus, $f_G + f_{aux}$ is sending $|f_G| - |f_{aux}|$ units of flow from s to t. Also, note that we can enforce $|f_{aux}| < L$ since $c(e_1) \geq f_G(e_1) > c'(e_1)$. This means that $f_G + f_{aux}$ is a valid flow for graph H which is at most L units of flow short of the maximum flow. So, similarly to part a, we can run Ford-Fulkerson on the residual graph $H_{f_G+f_{aux}}$ to find the maximum flow f_H in time $O((n+m)L)$.

Let $R = |f_G(e_1)| - c'(e_1)$. In order to complete our algorithm, it remains to show that we can also find f_{aux} in time $O((n+m)L)$. Consider the maximum flow f_G for graph G and let $e_1 = (u, v)$. We can see that $f_G(e_1)$ units from this flow go from t to v and through e_1 and continue from u to s. Doing so we have to be careful not to use the same edge in both the paths from t to v and from u to s. Therefore every time that we push flow from t to v we have to update the residual graph in order to be able to find a valid flow from u to s in the next step. One way to do this is to first find a flow f^1_{aux} from t to v that pushes R amount of flow. Then update the residual graph and then find a flow f^2_{aux} from u to s that pushes L amount of flow also. Now pushing flow R in the reverse direction of e_1 we create the wanted flow f_{aux}.

We can do every of the above steps by running a the augmenting procedure Ford-Fulkerson algorithm where we keep adding augmenting paths and cap the flow we will send $|f_{aux}|$ (i.e we might not saturate the last augmenting path we add). This will take time $O((n+m) \cdot |f_{aux}|) = O((n+m) \cdot L)$. Finally we have to run again Ford-Fulkerson in the end in order to make sure that we have found the optimal flow. This is done like in part a and costs $O((n+m)L)$. So the total running time is $O((n+m)L)$

Note that, the paths from s to u might not be disjoint from paths from v to t, but this is not a problem since the flow conservation constraints are satisfied on each node.

Quiz 2-5: [20 points] Linear Program for Matchings: Parts (a) – (c)

Given a simple undirected graph $G = (V, E)$ our goal is to see how we can use linear programs to compute maximum matchings in G. Note that G need not be bipartite. For any vertex $v \in V$ we define the neighborhood of v

$$\Gamma(v) = \{u \in V \mid \{u, v\} \in E\}$$

(a) [5 points] Using one variable x_e for every edge e, we formulate the maximum matching problem as the following integer linear program (ILP1)

$$\begin{aligned} & \max \sum_{e \in E} x_e \\ \text{(ILP1)}: \qquad & \textit{s.t.} \sum_{u \in \Gamma(v)} x_{\{u,v\}} = 1 \quad \forall v \in V \\ & x_e \in \{0, 1\} \end{aligned}$$

The formulation given above has a small mistake.

Correct the mistake in the above ILP (ILP1) to get another ILP (ILP2) that is the correct integer linear program formulation of the maximum matching problem. Also relax the integrality constraints of both (ILP1) and (ILP2) to get the corresponding linear programming relaxations (LP1) and (LP2) that might have fractional solutions.

Solution. The mistake is that the (ILP1) formulates the perfect matching problem instead of the maximum matching and therefore instead of equality to the constraints we should have inequality since it is possible that not every vertex has an edge in the maximum matching.

$$\begin{aligned} & \max \sum_{e \in E} x_e \\ \text{(ILP2)}: \qquad & \textit{s.t.} \sum_{u \in \Gamma(v)} x_{\{u,v\}} \leq 1 \quad \forall v \in V \\ & x_e \in \{0, 1\} \end{aligned}$$

The corresponding linear programming relaxations are

$$\begin{aligned} & \max \sum_{e \in E} x_e \\ \text{(LP1)}: \qquad & \textit{s.t.} \sum_{u \in \Gamma(v)} x_{\{u,v\}} = 1 \quad \forall v \in V \\ & 0 \leq x_e \leq 1 \end{aligned}$$

$$\max \sum_{e \in E} x_e$$

$$\text{(LP2)}: \quad \textit{s.t.} \sum_{u \in \Gamma(v)} x_{\{u,v\}} \leq 1 \quad \forall v \in V$$

$$0 \leq x_e \leq 1$$

For the rest of the problem, except for the last part, we will only consider *2-regular* graphs. This means that every vertex in G has degree exactly 2.

(b) [7 points] Given the optimal solution x_e^* of (LP1)[1] we define $E_1 = \{e \mid x_e^* \neq 1/2\}$ and $E_2 = \{e \mid x_e^* = 1/2\}$. Prove that both the graphs $G_1 = (V, E_1)$ and $G_2 = (V, E_2)$ consist of disjoint cycles.

Based on this show how you can change x^* to an optimal solution $\hat{x}$ such that for any $e \in E$, $\hat{x}_e \in \{0, 1/2, 1\}$.

Solution. First, because the graph is 2-regular, the graph is composed of disjoint cycles (as in the problem set).

Now, consider a traversal $v_1, v_2, ..., v_k, v_1$ of some arbitrary cycle. The first vertex in this cycle, v_1, is an endpoint of exactly two edges, $\{v_1, v_2\}$ and $\{v_k, v_1\}$. The solution must satisfy the LP1 constraints, so in particular for v_1

$$\sum_{u \in \Gamma v_1} x^*_{\{u,v_1\}} = 1$$

$$x^*_{\{v_1,v_2\}} + x^*_{\{v_k,v_1\}} = 1.$$

There are now three (mutually exclusive and) exhaustive causes:

- If $x^*_{\{v_1,v_2\}} = \frac{1}{2}$, then $x^*_{\{v_k,v_1\}} = 1 - x^*_{\{v_1,v_2\}} = \frac{1}{2}$ also. Now applying the same reasoning to the variables associated with the two edges incident on v_2, we see that $x^*_{\{v_2,v_3\}} = \frac{1}{2}$ as well. We can continue to do this for each vertex in the cycle to get that the entire cycle is composed of edges with associated solution variables with value $\frac{1}{2}$.
- If $x^*_{\{v_1,v_2\}} > \frac{1}{2}$, then $x^*_{\{v_k,v_1\}} = 1 - x^*_{\{v_1,v_2\}} < \frac{1}{2}$ also. We can apply similar reasoning to the variables associated with the two edges incident on v_2 to get that $x^*_{\{v_2,v_3\}} > \frac{1}{2}$, etc. So solution variables associated with the edges in this cycle alternate above and below $\frac{1}{2}$.
- If $x^*_{\{v_1,v_2\}} < \frac{1}{2}$, then $x^*_{\{v_k,v_1\}} = 1 - x^*_{\{v_1,v_2\}} > \frac{1}{2}$ also. As in the previous, we can see that the solution variables associated with the edges in this cycle alternate above and below $\frac{1}{2}$.

For cycles of the first type, we can leave the associated variables unchanged at $\frac{1}{2}$.

For the cycles of the second or third case, we can simply set the values below $\frac{1}{2}$ to 0 and the values above $\frac{1}{2}$ to 1.

[1] Although our goal is to deal with (LP2) you can show that it is the same to work with (LP1) in the case of 2-regular graphs.

Because the cycles are disjoint, each edge is contained by exactly one cycle, so this is a valid reassignment of the variables.

The constraints of LP1 are still satisfied. For any vertex in a cycle of the first type, its two edge values haven't changed. For any vertex in a cycle of the first type, its two edge values are now 0 and 1 so their sum is still 1.

The value of the objective function hasn't changed. For each edge in a cycle of the first type, its value hasn't changed. For each pair of consecutive edges in a cycle of the second type, the sum of those two values is still 1.

Now we turn our attention to general graphs that might not be 2-regular. For an optimal solution $\hat{x}$ we again define $E_1 = \{e \mid \hat{x}_e \neq 1/2\}$ and $E_2 = \{e \mid \hat{x}_e = 1/2\}$ and $G_1 = (V, E_1)$, $G_2 = (V, E_2)$.

(c) [8 points] We give you an unweighted graph $G = (V, E)$ and an optimal solution $\hat{x}$ of (LP2) such that for all $e \in E$, $\hat{x}_e \in \{0, 1/2, 1\}$. Also assume that G_2 contains **at most one odd cycle**. Provide an algorithm with running time $O(|V| + |E|)$ that given $\hat{x}$ finds a maximum matching in G.
Hint: As a first step prove there is an optimal matching that does not contain any edge $e \in E$ with $x_e = 0$ and contains all the edges $e \in E$ with $x_e = 1$.

Solution. Consider the graph $G_1' = (V, E_1')$ where $E_1' = \{e | \hat{x}_e = 1\}$. Also, notice that for any edge e such that value $x_e = 1$, all adjacent edges should have value 0 and for any edge e such that value $x_e = \frac{1}{2}$, either all adjacent edges should have value 0 or exactly one adjacent edge has value 1. That means that the graph G_1' is a matching and that G_2 consists of only even paths and arbitrary length cycles (using a similar argument to the one in part (b)). (Note that odd paths will not appear in G_2 for an optimal solution of LP2 since we could then increase the objective function by $\frac{1}{2}$ by setting the first and last edge to 1 and alternating 0's and 1's along the path, which is a contradiction.) Now, using an argument similar to the argument of part (b) we can see that we can turn any optimal solution x to an optimal solution where G_2 only has odd cycles. Indeed, consider an even path or cycle in G_2. We consider a numbering $e_1, \ldots, e_k$ of the edges of each even path starting from one endpoint of it (for an even cycle we can start anywhere). We set all odd numbered edges to 1 and all even numbered edges to 0. Then we update E_1'. This would preserve the value of the objective function and still satisfy the constraints of the LP. Therefore G_2 has only one odd cycle and all the other edges have value 0 or 1. Now we let M_2 be a maximum matching of G_2 that can be found in a similar way since G_2 is just an odd cycle. We set the following matching

$$M = E_1' \cup M_2$$

It is easy to see that M is a matching in G because it satisfies the contraints of (LP2) when we set $x_e = 1$ if and only if $e \in M$. Now the size of M, (which is also equal to the value of the objective function for the above assignment) is

$$|M| = \sum_{e|\hat{x}_e \neq 1/2} \hat{x}_e + |M_2|$$

But be optimal solution $\hat{x}$ has value

$$OPT = \sum_{e|\hat{x}_e \neq 1/2} \hat{x}_e + \sum_{e|\hat{x}_e = 1/2} \hat{x}_e = \sum_{e|\hat{x}_e \neq 1/2} \hat{x}_e + |M_2| + 1/2 = |M| + 1/2$$

Now assume that there was a matching M' with $|M'| > |M|$ and therefore $|M'| \geq |M| + 1$ if we define the solution $\tilde{x}$, such that

$$\tilde{x}_e = 1 \ \textit{if and only if} \ \ e \in M'$$

it would still satisfy the constraints and also the objective value of $\tilde{x}$ would be $|M'|$. However, from the above analysis we have that $|M| = OPT - 1/2$ and therefore $|M'| = OPT - 1/2 + 1 = OPT + 1/2$. But this contradicts with the assumption that $\hat{x}$ is an optimal solution. Therefore M is a maximum matching of G.

The running time of the algorithm is equal to the running time we need in order to convert possible even cycles and paths that have $\frac{1}{2}$-valued edges to matchings of 1-valued and then select all the 1-valued edges. This can be done by BFS/DFS in time $O(|V| + |E|)$.

Quiz 2-6: [20 points] Sum of Minima

We are given an array $A[1 \dots n]$ such that $A[i] \geq 0$ for any i. Our goal is to compute the quantity

$$B = \sum_{i=1}^{n} \sum_{j=i}^{n} \min_{i \leq k \leq j} A[k]$$

For example, if $A = [3, 1, 2]$ then $B = 9$. Provide an algorithm to compute B. For this problem the credit depends on the running time of your algorithm. Here is the list of the running times together with their credit.

Advice: Don't spend a lot of time on **iii.** or **iv.**. We suggest you to shoot for **ii.** and the rest of the exam and then return to try this problem.

i. [2 points] For an algorithm with running time $O(n^3)$.

ii. [15 points] For an algorithm with running time $O(n^2)$.

iii. [18 points] For an algorithm with running time $O(n \log n)$.

iv. [20 points] For an algorithm with running time $O(n)$.

Solution.

We give some sample solutions for each running time:

$O(n^3)$ **running time:**

We iterate through all possible intervals $[i, j]$, compute the minimum in each of them and add them. Since there are $O(n^2)$ intervals and finding the minimum takes $O(n)$ time, the total running time is $O(n^3)$.

$O(n^2)$ **running time:** We will fill two 2D matrices M, S defing as follows: Each entry of M is the minimum element of the subarray $A[i \dots j]$ and each entry of S is the sum of minima of the subarray $A[i \dots j]$.

We will start by filling in the diagonal entries $M_{i,i}, S_{i,i}$ and continue with the other diagonals below the main diagonal. In particular, we set $M_{i,i} = S_{i,i} = A[i]$ since this is the only element in the subarray. For the entries of the second diagonal, we use the following formulas:

$$M_{i,i+1} = \min\{M_{i,i}, M_{i+1,i+1}\}$$

$$S_{i,i+1} = S_{i,i} + S_{i+1,i+1} + M_{i,i+1}$$

For the 3rd diagonal onwards, we use the following recurrence relations:

$$M_{i,j} = \min\{M_{i+1,j}, M_{i,j-1}\}$$

$$S_{i,j} = S_{i,j-1} + S_{i+1,j} - S_{i+1,j-1} + M_{i,j}$$

Algorithm 1 SUM OF MINIMA(A)

```
b ← {0, n + 1}
sum ← 0
B ← the sorted order of i's in {1, ..., n} based on the vlaue A[i]
for t = B[1], B[2], ..., B[n] do
    t_l ← b.lower-bound(t)
    t_r ← b.upper-bound(t)
    sum ← sum + (t − t_l)(t_r − t)A[t]
end for
return sum
```

Note that in order to compute each entry $M_{i,j}$, we only need entries of M that correspond to smaller intervals and in order to compute each entry $S_{i,j}$, we only need entries of S that correspond to smaller intervals and the entry $M_{i,j}$. So, if we compute each diagonal one by one starting from the main we can fill in each entry of M and S in constant time. Since there are $O(n^2)$ entries we have to compute, the running time will be $O(n^2)$. Also, by definition of the matrix S, we have that $B = S_{1,n}$ is the answer to the problem.

$O(n \log n)$ **running time:** Without loss of generality assume $A[i]$'s are distinct. If not, we can use the standard technique: replace $A[i]$'s by $(A[i], i)$ to make all the elements distinct. To compare two pairs, first we compare the first entry, if they were equal we consider the second entry. Now, the minimum of each interval is unique and well-defined.

Let $M[t]$ be the number of intervals $[i, j]$ that contains t such that $A[t]$ in minimum in $A[i, \ldots, j]$. This will gives us another way to calculate the sum the sum, since

$$\sum_{i=1}^{n} \sum_{j=1}^{n} \min_{i \le k \le j} A[k] = \sum_{i=1}^{n} M[i]A[i].$$

Assume $t_l < t$ (and $t_r > t$) is the largest (and smallest) index such that $A[t_l] < A[t]$ ($A[t_r] < A[t]$). If all the entries on the left (right) of $A[t]$ are larger than it, then let $t_l = 0$ (t_r = n+1). By our definition, it is not hard to see that $A[t]$ is the minimum of $A[t_l+1, \ldots, t_r-1]$ and any sub-interval of it that contains t Also, it is not the minimum of any other interval. Thus, $M[t] = (t - t_l)(t_r - t)$.

In order to find t_l and t_r, we use a binary search tree called b which is empty initially. We sort the array $(A[i], i)$ in a increasing order and insert i in b at each step. Before inserting $(A[t], t)$, t_r and t_l are already inserted in b (because $A[t_r], A[t_l] < A[t]$) and no elements in $[t_l + 1, t_r - 1]$ is inserted. Thus, by we can find the lower bound and the upper bound of t in the b in $O(\log n)$.

We use Algorithm 1 to calculate $M[t]$. We spend $O(n \log n)$ time at the begining to sort. Also, we find the t_l and t_r in $O(\log n)$ time. Thus, the total running time is $O(n \log n)$

Algorithm 2 SUM OF MINIMA(A)

```
1: stack ← ∅
2: M[0] ← 0
3: sum ← 0
4: for t = 1, ..., n do
5:    while stack.is-empty() = false and A[stack.top()] ≥ A[t] do
6:       stack.pop()
7:    end while
8:    if stack.is-empty() = true then
9:       p_t ← 0
10:   else
11:      p_t ← stack.top()
12:   end if
13:   stack.push(t)
14:   M[t] ← (t − p_t)A[t] + M[p_t]
15:   sum ← sum + M[t]
16: end for
17: return sum
```

$O(n)$ **running time:** We define $M[t]$ to be the sum of minimum values of the intervals $[i, t]$ for any $i \leq t$. More precisely,

$$M[t] = \sum_{i=1}^{t} \min_{i \leq k \leq t} A[k].$$

It is clear that the final answer is $\sum_{j=1}^{n} M[j]$. In order to initialize our array, we can define $M[0]$ to be zero. Now, we can assume we have the values of $M[1], M[2], \ldots, M[t-1]$ and we want to find $M[t]$. We define p_t to be the largest index of such that $A[p_t] < A[t]$. If the p_t does not exist (i.e. $A[t] \leq A[1], \ldots, A[t-1]$), let p_t be zero. Now, we claim

$$M[t] = (t - p_t)A[t] + M[p_t].$$

To prove the claim, we consider two cases. First, by definition of p_t the minimum of $A[i, \ldots, t]$ is A[t] for any i such that $p_t < i \leq t$. Since we have $t - p_t$ of such i then we have $\sum_{i=p_t+1}^{t} \min_{i \leq k \leq t} = (t - p_t)A[t]$. Second, if $p_t \neq 0$, for any $i \leq p_t$, the minimum of $A[i, \ldots, t]$ is equal to the minimum of $A[i, \ldots, p_t]$, because all the entries in $A[p_t + 1, \ldots, t]$ are greater than $A[p_t]$ and does not change the value of minimum. Thus, $\sum_{i=1}^{p_t} \min_{i \leq k \leq t} A[k] = \sum_{i=1}^{p_t} \min_{i \leq k \leq p_t} A[k] = M[p_t]$. This is also consistent when $p_t = 0$.

We use Algorithm 2to find p_t and updating M.

Assume we found p_t for a fixed t based on the above algorithm. We want to show that all the elements in $A[p_t + 1, ..., t - 1]$ was greater than $A[t]$. By contradiction assume there exists a $t_1 > p_t$ such that $A[t_1] < A[t]$. Since we picked p_t which is less than t_1, it means

that t_1 has to be popped from the stack before t. If t_1 is popped form the stack, then another there exists another $t_2 > t_1$ such that $A[t_2] < A[t_1] < A[t]$ and we can repeat this argument infinitely many times. However, the number of elements between p_t and t are finite. Thus, there are not infinitely many t_i's to repeat this argument. Thus, we reach a contradiction and such t_1 does not exist from the beginning. Hence, the algorithm pick the write p_t and output the write answer.

To analyze this algorithm, note that each element will be pushed in stack once. thus, the total number of "pop" operation in the "while loop" is $O(n)$. Thus, the total running time of the algorithm is $O(n)$.