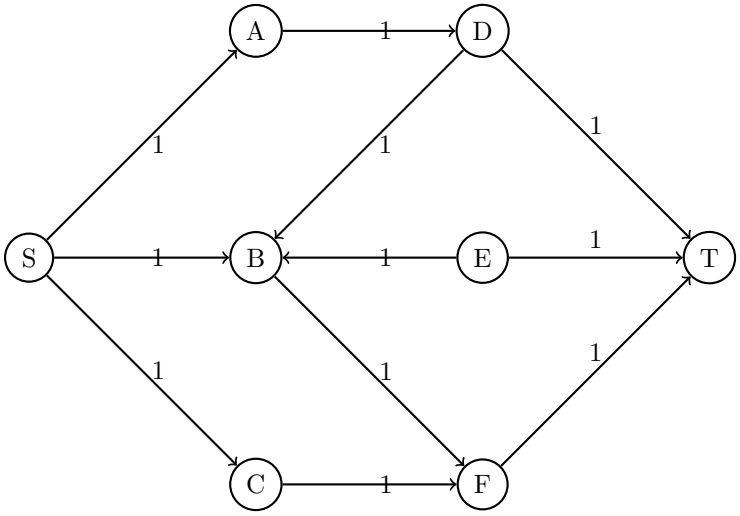
Final Solutions

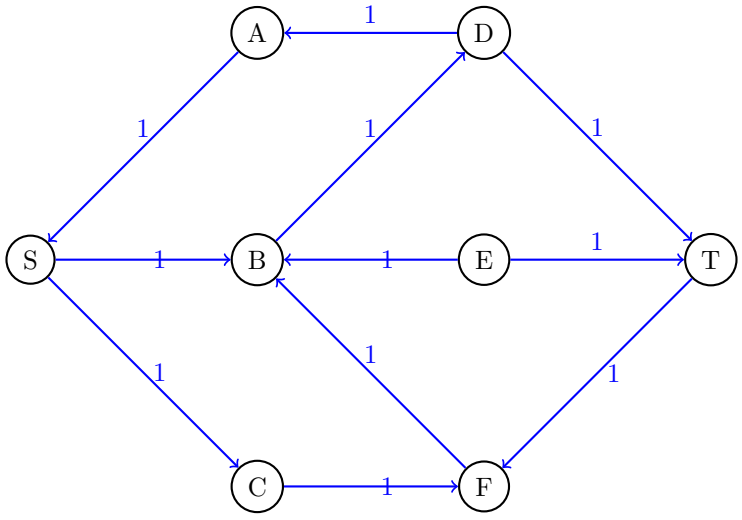
1 Maximum Flow (6 points)

Consider the following directed graph with all edge capacities equal to 1.



In the first step of Maximum-Flow algorithm, we increase the flow along $S \to A \to D \to B \to F \to T$ by one unit.

1. Draw the residual graph after this step.



2. What happens next in the execution of Max-Flow algorithm? (the algorithm does not necessarily run for three steps)

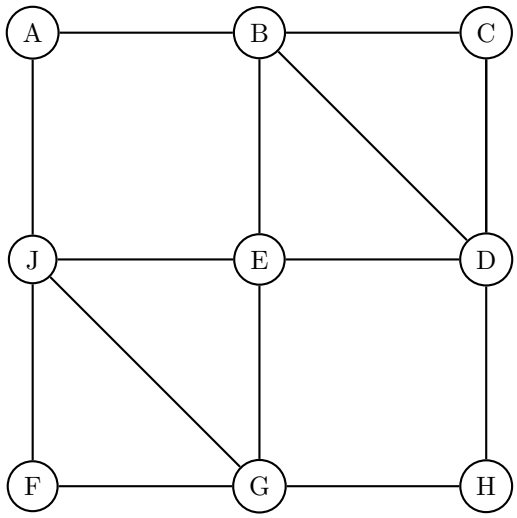
   - Send [1] unit(s) of flow on path $S \rightarrow$ [$B \rightarrow D$] $\rightarrow T$
     S, C, F, B, D, T also works
   - Send [ ] unit(s) of flow on path $S \rightarrow$ [ ] $\rightarrow T$
   - Send [ ] unit(s) of flow on path $S \rightarrow$ [ ] $\rightarrow T$

3. What is the minimum $S - T$ cut in the graph?

   - S-side of the partition $= \{S,$ [S, C, F, B] $\}$
   - T-side of the partition $= \{T,$ [A, D, E, T] $\}$
     A can be on either side of the cut

2 Minimum Spanning Tree (6 points)

Consider the following graph, and the list of edges in increasing order of their weights.



1. (A, B)
2. (B, C)
3. (C, D)
4. (B, E)
5. (B, D)
6. (A, J)
7. (F, J)
8. (D, E)
9. (E, G)
10. (D, H)
11. (G, J)
12. (G, H)
13. (E, J)
14. (F, G)

Run Kruskal's algorithm and Prim's algorithm (starting at node E for the latter). Which are the second, fourth, and seventh edges that are added to the resultant MSTs, in each of the two algorithms?

	Kruskal's Algorithm	Prim's Algorithm
Step 2	(B,C)	(A,B)
Step 4	(B,E)	(C,D)
Step 7	(E,G)	(E,G)

3 Vertex Cover (6 points)

Suppose $x_1 = \frac{3}{4}, x_2 = \frac{1}{3}, x_3 = \frac{2}{3}, x_4 = \frac{2}{3}, x_5 = \frac{1}{4}, x_6 = \frac{2}{3}, x_7 = \frac{1}{3}$ be the optimal solution to the linear programming relaxation of vertex cover on a graph G (all vertex weights are 1).

1. Choose the pairs of vertices that are NOT edges in the graph. Bubble in your answers. [Note: This part will be graded automatically. Please mark your answer clearly.]

   ◯ (1,2)
   ◯ (2,3)
   ◯ (2,4)
   ◯ (4,5)
   ◯ (3,6)
   ◯ (6,7)

   (4,5). In LP formulation of the vertex cover, we need to make sure for each two vertices i and j, $x_i + x_j \geq 1$. So if we check this constraint to all edges above, (4, 5) is the only one that the LP solutions do not satisfy.

2. The size of the smallest vertex cover in G is at least [4]

   4. LP solution should be the lower bound of the optimal solution. If we sum of all x_i, it sums up to $3\frac{2}{3}$. Therefore the x vertex cover is 4.

3. Choose the vertices that belong to the vertex cover output by the approximation algorithm. Bubble in your answers. [Note: This part will be graded automatically. Please mark your answer clearly.]

   ◯ 1
   ◯ 2
   ◯ 3
   ◯ 4
   ◯ 5
   ◯ 6
   ◯ 7

   Approximation algorithm says that you need to choose all the vertices that have value greater than 1/2. In this case, they are x_1, x_3, x_4, x_6.

4. The algorithm yields a [1] approximation on this graph.

   From part b and c, we see the algorithm we have, gives answer of 4 vertices, and the optimal solution is at least 4 vertices. In this case, we get the optimal solution. It is 1-approximation on the graph.

4 Zero-Sum Games (3 points)

Consider a zero-sum game given by the following matrix (indicating the payoffs to the row player)

	C_1	C_2	C_3
R_1	1	2	3
R_2	2	2	1
R_3	1	4	2

Suppose the column player goes second and has fixed the following probabilistic strategy:

C_1	C_2	C_3
0.3	0.2	0.5

what is the best strategy for the row player going first?

R_1	R_2	R_3
1.0	0.0	0.0

The value the row player gains from choosing R_1 is 2.2, from R_2 is 1.5, and R_3 is 2.1, so the row player should choose R_1 always.

5 Linear Programming (6 points)

1. Write the dual of the following linear program.

$$\begin{aligned} \max \quad & 3x_1 + x_2 + 4x_3 + 4x_4 + x_5 \\ & x_1 + x_2 + x_3 + x_4 + x_5 \leq 2 \\ & x_1 - x_2 + 2x_3 + 3x_4 \leq 1 \\ & x_1, x_2, x_3, x_4, x_5 \geq 0 \end{aligned} \tag{1}$$

   The dual is as follows:

$$\begin{aligned} \min \quad & 2y_1 + y_2 \\ & y_1 + y_2 \geq 3 \\ & y_1 - y_2 \geq 1 \\ & y_1 + 2y_2 \geq 4 \\ & y_1 + 3y_2 \geq 4 \\ & y_1 \geq 1 \\ & y_1, y_2 \geq 0 \end{aligned} \tag{2}$$

2. Prove that the optimum of the linear program given in part 1 is at most 5.

   We note that if we set $y_1 = 2$ and $y_2 = 1$, we satisfy all the constraints of the dual linear program, and its objective attains a value of $2(2) + (1) = 5$. Thus, the optimal (minimum) value of the dual program is at most 5. Because the optimal (maximum) value of the primal can be at most the minimum of the dual, it must also be at most 5.

6 Factoring Numbers via Circuit SAT (5 points)

Let $n = 1010001010100010101111110000010101010111111111111111111000000000011100000000111$.

Describe a circuit C (in at most two sentences), such that solving the CircuitSAT problem on the instance C will yield a factor p dividing n.

To check if p is a factor of n, we simply need to check if (integer division/multiplication) $n/p * p == n$. We can create a circuit that implements the division and multiplication, with n hardwired to be the number provided. Both division and multiplication are polynomial in the bits of n: there can only be as many operations as there are bits in n. Then $n/p * p$ and n are fed into the final gate, an equal/XNOR gate, which outputs a 1 if p is a factor of n. The inputs to the circuit are the bits in p. Solving this CircuitSAT problem is equivalent to the question "What input would cause this circuit to output 1?", which would find the factor p.

This problem shows why CircuitSAT is NP-complete. Factoring is an incredibly hard problem in CS (and the backbone of many encryption schemas), and CircuitSAT can solve it along with *any* problem with a polynomial time verifier. You can build a polynomial size circuit to do any polynomial time algorithm, so if you solve CircuitSAT then you can solve any problem in NP.

7 Find the satisfying one (5 points)

You are given a 3-SAT formula Φ and two assignment sets x_1, x_2 such that one of the assignment sets satisfies Φ, while the other satisfies at most 90% of the clauses in Φ.

Describe an $O(1)$-time algorithm that finds the satisfying assignment among x_1, x_2 with probability 0.99.

Main Idea:

We randomly choose one assignment set. Then randomly pick k clauses and test on the assignment set. If any clause doesn't satisfy, we pick the other assignment set. Otherwise, if all clauses are satisfied, we pick this set.

Prove:

We set our k in the following way:

1. You can also use Chernoff bound to approach this problem. Set the error to be 0.1 and the probability bounded by 0.01. $k = \lceil \frac{1}{2 * 0.1^2} \log_e \frac{2}{0.01} \rceil$

2. Asymptotically, we have infinite number of clauses. Therefore, since we just try a little portion of the clauses, we can assume that in the wrong assignments set, each clauses is satisfied with probability $P(\text{One clause satisfied}|\text{Wrong set}) \leq 0.9$

If we randomly pick k clauses from one set and check if all the k clauses are satisfied. Then:
$P(\text{k clauses all satisfied}|\text{Wrong set}) = P(\text{One clause satisfied}|\text{Wrong set})^k \leq 0.9^k$

$$P(\text{Correct set}|\text{k clauses all satisfied})$$
$$= \frac{\frac{1}{2} * P(\text{Correct set}|\text{k clauses all satisfied})}{\frac{1}{2} * P(\text{Correct set}|\text{k clauses all satisfied}) + \frac{1}{2} * P(\text{Wrong set}|\text{k clauses all satisfied})}$$
$$= \frac{1}{1 + 0.9^k}$$

In the boundary case, we set the $P(\text{Correct set}|\text{k clauses all satisfied})$ to 0.99.
$\frac{1}{1 + 0.9^k} = 0.99$
$k = \lceil \log_{0.9}(\frac{1}{0.09} - 1) \rceil = 44$

8 Updating Distances (8 points)

We have a directed graph $G = (V, E)$, where each edge (u, v) has a length $\ell(u, v)$ that is a positive integer. Let n denote the number of vertices in G. In other words, $n = \|V\|$. Suppose we have previously computed a $n \times n$ matrix $d[\cdot, \cdot]$, where for each pair of vertices $(u, v) \in V$, $d[u, v]$ stores the length of the shortest path from u to v in G. The $d[\cdot, \cdot]$ matrix is provided to you. Now we add a single edge (a, b) to get the graph $G' = (V; E')$, where $E = E \cup \{(a, b)\}$. Let $\ell(a, b)$ denote the the length of the new edge. Your job is to compute a new distance matrix $d'[\cdot, \cdot]$, which should be filled in so that $d'[u, v]$ holds the length of the shortest path from u to v in G', for each $u, v \in V$.

1. Write a concise and efficient algorithm to fill in the $d'[\cdot, \cdot]$ matrix. You should not need more than about 3 lines of pseudocode. (psuedocode only)

   for $i = 1, \ldots, n$:
     for $j = 1, \ldots, n$:
       $d'[i, j] = \min(d[i, j], d[i, a] + l(a, b) + d[b, j])$

2. What is the run-time of your algorithm?

   Run-time of the algorithm is $O(n^2)$

9 Finding Bridges (20 points)

A bridge of an undirected graph G is an edge whose removal disconnects G.

1. An edge is a *bridge* if and only if it is not part of any | Cycle |

2. Suppose we perform a DFS of the graph G then a *bridge* can be (bubble all possibilities): [Note: This part will be graded automatically. Please mark your answer clearly.]

   ◯ Tree edge

   ◯ Back edge

   Tree edge.

3. Suppose we perform a DFS of the graph G starting from s. At some point during the execution of the DFS, let us suppose the current node is v_t. Suppose the path from the root of DFS tree to the current node v_t is given by,

   $$s = v_0 \to v_1 \to v_2 \ldots \to v_{t-1} \to v_t$$

   and the DFS encounters a back edge $v_t \to v_i$ for some $i < t$. Which edges of the graph are definitely not bridges?

   All edges in the cycle: $(v_i \to v_{i+1}), \ldots, (v_{t-1} \to v_t)$.

4. Use the hints above to design an $O(|E|)$ time modification of DFS to find all the bridges in an undirected graph G. For simplicity, let us assume that assume that G is connected.

   Briefly state your main idea and fill in the blanks in the following pseudocode. You don't need to use all the lines that we provide.

   Main Idea:

   We know an edge is not a bridge if it is part of a cycle; we keep track of this in the boolean array $isBridge$. In our DFS traversal, once we encounter a back-edge (u, v), we know it cannot be a bridge. We also need to propagate this information back to the other earlier edges in the cycle.

   Staff Solution: We check the direct ancestor of u with this information in postvisit(u). If a node is part of a cycle, it is either the head of the cycle or its direct ancestor is part of the cycle (depth[earliest[v]] < depth[u]). We do this by keeping track of the "earliest" node in the cycle containing the current node u. We also update the ancestor v with information about the earliest node in the cycle ($earliest[u]$).

   There are other possible solutions that handle the cycle edges through chains of treeEdge[] calls directly instead of using recursive postvisit() calls, but many did not fit the $O(|E|)$ time requirement.

   Common mistakes included mistakes with setting isBridge[] to True for edges in the cycle or forgetting to set isBridge[] to True for edges not in cycles, mistaking the ordering of explore() and postvisit() calls, not taking the minimum of depths when updating earliest[] values for edges in the cycle, and updating all neighboring edges' earliest[] values in postvisit() rather than just the ancestor's.

Pseudocode:

Let

$depth[u] =$ depth of node u in DFS tree
$depth =$ current depth of DFS
$visited[u] =$ boolean indicating whether visited u already
$isBridge[u, v] =$ True if (u, v) is a bridge in the graph, and initialized to False for every edge at the beginning
$earliest[u] =$ Earliest ancestor that is part of some cycle with u, initialized to u.
$treeEdge[u] =$ Tree edge leading to u

procedure EXPLORE(u):

for each edge$(u, v) \in E$ **do**

if visited[v] = True **then**

 # (isBridge[u,v] = False)
if depth[v] $\leq$ depth[earliest[u]] **then**
earliest[u] = v
end if
else
treeEdge[v] = (u,v)
Previsit(v)
Explore(v)
Postvisit(v)
end if
end for
end procedure

procedure PREVISIT(u):

$depth = depth + 1$
depth[u] = depth
visited[u] = True
end procedure

procedure POSTVISIT(u):

$depth = depth - 1$
 # Look only at tree edge from node u to direct ancestor w
(u, w) = treeEdge[u]
 # Check if w is part of some cycle with u (isBridge[u,w] = False)
if depth[earliest[u]] $\leq$ depth[w] **then**
 # Update 'earliest' entry for w if u has an earlier ancestor, check is necessary in case of multiple cycles
if depth[earliest[u]] $<$ depth[earliest[w]] **then**
earliest[w] = earliest[u]
end if
else
isBridge[u,w] = True

```
end if
end procedure
```

10 Gambling (15 points)

You walk into a casino. You have M dollars of money, and want to play exactly n rounds of games. Let us call it a *success* if at the end of the n games, you have exactly $2M$ dollars (no more, and no less).

For each of the n rounds, you can choose to play either Game A or Game B. Game A costs \$1 to play, and returns \$2 with probability 0.6 (and \$0 with probability 0.4). Game B costs \$3 and returns \$15 with probability 0.2 (and \$0 with probability 0.8).

(The returns do not include the cost, so if you play and win Game A, your net gain is \$1.)

We will now design a dynamic programming algorithm to compute the probability of *success* of the optimal strategy.

Define the subproblem as the following:

$T[m, \ell]$ = Optimal strategy's probability of *success*, if you have m dollars at the end of ℓ games

1. Base cases:

   (Note: A lonely m or a ℓ represents this can be any value)

   Condition for *success*:

   $$T[m = 2M, \ell = n] = 1$$

   Playing n games, but not having exactly $\$2M$ at the end:

   $$T[m \neq 2M, \ell = n] = 0$$

   Having zero money at any point means you can't play any more games (assum M is positive):

   $$T[m = 0, \ell] = 0$$

   The following was not required for full points, but good to have:

   $$T[m < M - 3n, \ell] = 0$$

   (the above is the case where you lost as much money as possible, and is helpful for bounding the runtime)

   Playing more than n games is not okay:

   $$T[m, \ell > n] = 0$$

   You can only win so much money (also helpful in bounding the runtime):

   $$T[m > M + 12n, \ell] = 0$$

2. Recurrence relation:

   If $m < 3$, we can only play game A, so we have:

   $$T[m, \ell] = 0.6 \cdot T[m + 1, \ell + 1] + 0.4 \cdot T[m - 1, \ell + 1]$$

(some answers put the above in the base case section, which is OK)

Otherwise,

$$T[m, \ell] = \max\{0.6 \cdot T[m+1, \ell+1] + 0.4 \cdot T[m-1, \ell+1], 0.2 \cdot T[m+12, \ell+1] + 0.8 \cdot T[m-3, \ell+1]\}$$

Having an optimal strategy means you always take the option that has a higher chance of giving you *success*. For either option, we can compute the probability of *success* in terms of two possible outcomes.

3. The run-time of the algorithm is $\Theta($ $\min\{15n, M+12n\} \cdot n$ $)$

How many subproblems are there? We need to see how many possible configurations of (m, ℓ) there can be.

First, ℓ can range from 0 to n, so this contributes an n multiplicative factor. Answers using ℓ instead of n were also accepted.

Now consider money. The greatest amount of money possible is if we win Game B n times: $M + 12n$. The least amount of money possible is either:

(a) $M - 3n$. We lost Game B n times.

(b) 0. We lost money until we cannot play anymore.

So the number of configurations for money is $\min\{\text{case (a)}, \text{case (b)}\} = \min\{15n, M+12n\}$.

Putting these two together, this is:

$$\Theta(\min\{15n, M+12n\} \cdot n)$$

We don't know if M or n dominates, so this doesn't simplify further. Answers along these lines were accepted:

- $\Theta(15n^2)$.
- $\Theta((M+12n)n)$

$\Theta(Mn)$ is not exactly correct, because this implies there were $2M$ possible values for money, and not on the order of n or $M+n$. For example, suppose if someone has $2M+n$ dollars, and then loses Game B $\frac{n}{3}$ times to end up with $2M$ at the end.

11 Roadside Assistance (20 points)

You are the CEO of a towing company that serves a network of roads connecting n cities. The road network is given by an undirected graph $G = (V, E)$ where $V = \{1, \ldots, n\}$ is the set of cities, and E denotes the set of roads connecting pairs of cities in V.

On each road $(i, j) \in E$, there are w_{ij} accidents that occur each day, which need road-side assistance. An accident occuring on road (i, j) can only be serviced by a tow-truck from city i or city j. Each accident needs exactly one tow-truck for assistance.

At each city i, the company parks t_i tow-trucks.

1. (10 points) Describe a polynomial-time algorithm to determine whether the company can service all the accidents. If so, determine how the company should service the accidents, i.e., For each road $(i, j) \in E$, the algorithm must determine how many of the w_{ij} accidents on the road are assisted by trucks from i, and how many by trucks from j?

   Describe the main idea of the algorithm precisely, proof of correctness is not needed.

   We will reduce this problem to max flow.

   - Create a new graph $G' = (V', E')$. We will first add all vertices v_k from V to V'.
   - Add source vertex s and sink vertex t.
   - For each edge $(i, j) \in E$, we create new vertices x_{ij} and add it to V'.
   - For each vertex $x_{ij} \in V'$, we add directed edges (v_i, x_{ij}) and (v_j, x_{ij}) to E' with capacities t_i and t_j respectively. We can service at most t_i accidents from city i for road (i, j), and so forth.
   - There will also be a directed edge to the sink (x_{ij}, t) with capacity w_{ij}. This allows us to service all accidents for road (i, j).
   - For all original vertices $v_k \in V'$, we add directed edges from the source (s, v_k) with capacity t_k to E'. This will ensure that we cannot service more than t_i accidents from city i.
   - Run max flow on new graph $G' = (V', E')$.

   If the incoming edges to t are fully saturated, then there is a feasible solution to our roadside assistance problem. We can look at the flow values of individual edges (v_k, x_{ik}) to determine how many trucks we service from city k for road (i, k).

2. (10 points) The company realized that it is paying too much parking fees for the trucks at the cities. Parking a truck in city i costs c_i.

   The company would like to rearrange all the trucks, so as to minimize the total parking costs, while still being able to assist all the accidents on every road.

   Write a linear program to determine how to rearrange the trucks, so that they can still assist all the w_{ij} accidents on each road (i, j), but minimize the total parking cost.

   (a) What are the variables of the linear program?

   - Let x_{ij} be the number of trucks parked at city i that will service the road (i, j).
   - Let p_i be the number of trucks parked at city i.

   (b) What is the objective function?

   We want to minimize total parking costs.

$$\min \quad \sum_{i \in V} c_i \sum_{(i,j) \in E} x_{ij}$$

(c) What are the constraints?

$$\sum_{i \in V} \sum_{(i,j) \in E} x_{ij} = \sum_{i \in V} t_i \quad \text{Total number of trucks}$$

$$\forall i \in V, \sum_{(i,j) \in E} x_{ij} = p_{ij} \quad \text{only cars parked at } i \text{ can service the edges}$$

$$\forall (i,j) \in E \quad x_{ij} + x_{ji} \geq w_{ij} \quad \text{all accidents on edge } (i,j) \text{ are serviced}$$

$$\forall i \in V \quad p_i \geq 0 \quad \text{Nonnegativity}$$

$$\forall i \in V, (i,j) \in E \quad x_{ij} \geq 0 \quad \text{Nonnegativity}$$

Alternate Solution

$$\min \quad \sum_{i \in V} c_i p_i$$

Subject to

$$\sum_{i \in V} p_i = \sum_{i \in V} t_i \quad \text{Total number of trucks}$$

$$\forall i \in V, \sum_{(i,j) \in E} x_{ij} \leq p_{ij} \quad \text{only cars parked at } i \text{ can service the edges}$$

$$\forall (i,j) \in E \quad x_{ij} + x_{ji} \geq w_{ij} \quad \text{all accidents on edge } (i,j) \text{ are serviced}$$

$$\forall i \in V \quad p_i \geq 0 \quad \text{Nonnegativity}$$

$$\forall i \in V, (i,j) \in E \quad x_{ij} \geq 0 \quad \text{Nonnegativity}$$

12 Complete the sentences: *(26 points)*

When asked for a bound, always give the tightest bound possible.

1. The number of strongly connected components in an n vertex DAG is at least [n].
2. Suppose $E[i,j]$ is the ij^{th} entry of the table in the dynamic programming based algorithm for edit distance, then $E[5,6]$ can be computed using the entries: [E[4,6], E[4,5], E[5,5]]
3. Every directed graph can be decomposed in to a [DAG of strongly connected components]
4. The solution to the recurrence $T(n) = 8T(n/4)+O(n^2)$ is [$T(n) = \Theta(n^2)$] .
5. Suppose $T(2^n) = T(n) + 1$ and $T(1) = 1$ then $T(n) =$ [$\Theta(log^*(n))$] .
6. [Cross] edges occur in the DFS tree only if the graph is directed.
7. Running DFS starting from a node in a [sink SCC] yields a strongly connected component of the graph.
8. Minimum spanning tree of a graph never contains the heaviest edge in every [cycle]
9. Minimum spanning tree of a graph always contains the lightest edge in every [cut]
10. Suppose a hash function $h : \{0, 1, \ldots, p-1\} \to \{0, 1, \ldots, p-1\}$ chosen from a universal hash family then $\Pr[h(2) = 2 \cdot h(1) \mod p] =$ [$1/p$]
11. There are 8 symbols $\{A, B, C, D, E, F, G, H\}$ all of whose frequencies are within the range $[\frac{1}{8} - 10^{-3}, \frac{1}{8} + 10^{-3}]$. The Huffman encoding of these symbols will consist of strings [$\{000, 001, 010, 011, 100, 101, 110, 111\}$] in some order.
12. The cost of every TSP tour in a graph is always [greater] than the cost of a minimum spanning tree. (positive weights, complete graph)
13. Two polynomials $p(x), q(x)$ given in the (point,value) representation can be multiplied in time [$\Theta(n)$]. (both are degree at most n)

13 True or false? (12 points)

Bubble in the right answer. No explanation needed. No points will be subtracted for wrong answers, so guess all you want! This part will be graded automatically. Please mark your answer clearly.

1. Dijkstra's algorithm does not work on every DAG with negative edge weights.

   ◯ True

   ◯ False

   True, consider a DAG on 4 nodes with diamond shape with weights $\{1, 1\}$ along one path and weights $\{2, -2\}$ along the other, from source to target.

2. On a graph with integer capacities, the size of the minimum cut is integral.

   ◯ True

   ◯ False

   True, recall that the value of the max flow of a graph with integer capacities must be integral.

3. On a graph with only integer capacities, the total maximum flow between any chosen pair s, t can be non-integral.

   ◯ True

   ◯ False

   False, it will always be an integer, as in the previous question.

4. If a linear program has an integral optimal value then it has a unique solution.

   ◯ True

   ◯ False

   False, consider for example $\max 0 : -1 \leq x \leq 1$.

5. The set of all functions from $\{0, 1, \ldots, p-1\}$ to $\{0, 1, \ldots, p-1\}$ is a universal hash family.

   ◯ True

   ◯ False

   True.

6. Minimum Vertex Cover is polynomial-time solvable on a tree.

   ◯ True

◯ False

True, a dynamic programming algorithm for this task was covered in section.

7. Doubling the capacities of all edges of a minimum cut in a graph G, doubles the maximum flow.

◯ True

◯ False

False, the second-best minimum cut may have value less than double the max flow.

8. Simplex algorithm can be used to solve linear programs in polynomial time.

◯ True

◯ False

False, simplex is an exponential time algorithm.

9. The number of sub-problems in computing edit distance between two strings $x[1 \dots m]$ and $y[1 \dots n]$ is $O(mn)$.

◯ True

◯ False

True, recall we define $E[i, j]$ for $i = 1 \dots m, j = 1 \dots n$.

10. The value of the edit distance between two strings $x[1, \dots, m]$ and $y[1, \dots, n]$ can be computed using $O(m + n)$-space.

◯ True

◯ False

True, it is possible to compute edit distance only tracking entries along the preceding diagonal.

11. For any proof, if a cheating verifier cannot recover the prover's secret witness from an interactive proof then the proof is zero-knowledge.

◯ True

◯ False

False! The converse of the statement is true but this statement is false. In particular, the requirement for a proof to be zero-knowledge is stronger — namely, the cheating verifier learns nothing about the prover's secret (e.g. even the first bit of the prover's secret remains hidden from the verifier).

12. Minimum Vertex Cover is known to be an NP problem.

◯ True

◯ False

False, there is no way to efficiently check if a vertex cover is the smallest one.

14 True or False or Maybe..? (12 points)

In the remaining questions there are four possible answers: (1) True (T); (2) False (F); (3) True if and only if $\mathbf{P} = \mathbf{NP}$ *(=); (3) True if and only if* $\mathbf{P} \neq \mathbf{NP}$ *(≠).*

Bubble in the most appropriate one. **Note:** *By "reduction" in this exam it is always meant "polynomial-time reduction."*
This part will be graded automatically. Please mark your answer clearly.

13. There is a reduction from Bipartite Matching to Independent Set.
    ◯ True
    ◯ False
    ◯ =
    ◯ ≠
    T

14. There is a reduction from Independent Set to Maximum Flow.
    ◯ True
    ◯ False
    ◯ =
    ◯ ≠
    =

15. There is a polynomial-time algorithm for 3-SAT.
    ◯ True
    ◯ False
    ◯ =
    ◯ ≠
    =

16. There is a reduction from Factoring to Rudrata Path.
    ◯ True
    ◯ False

◯ =

◯ ≠

T

17. Every problem in **NP** can be written as an integer linear program.

◯ True

◯ False

◯ =

◯ ≠

T

18. Minimum Spanning Tree is an NP problem.

◯ True

◯ False

◯ =

◯ ≠

T

15 NP-completeness *(20 points)*

Note: By "reduction" in this exam it is always meant "polynomial-time reduction." For the reductions in Problem **??** mention the problem you are using, direction and construction of the reduction). Also, when you are asked to show that a problem is **NP**-complete, no need to show that it is in **NP**, unless asked to do so.

You may assume that the following problems are known to be **NP**-hard.

- Rudrata Path or Hamiltonian Path
- Hamiltonian Cycle
- Vertex Cover
- Independent Set
- 3-SAT
- CircuitSAT
- Integer Programming
- Clique
- 3D-Matching
- 4-SAT
- 3-Coloring

(8 points) 1) Balanced 3SAT: In the balanced 3-SAT formula, the input is a 3-SAT formula and the goal is to find a satisfying assignment $x = (x_1, \ldots, x_n)$ such that exactly half the variables are assigned 0, and half assigned 1.

Proof: *We will reduce the problem ...* *to the problem ...*

Given an instance Φ *of the problem ...* *we construct an instance* Ψ *of the problem ...*

as follows ...

The proof that this is a valid reduction is as follows:

■

We will reduce the problem 3SAT to the problem Balanced-3SAT

Given an instance Φ of the problem 3SAT, we construct an instance Ψ of the problem Balanced-3SAT

as follows:

For each clause $x_i \vee x_j \vee x_k$ is Φ, we put it and add an additional clause $\bar{y_i} \vee \bar{y_j} \vee \bar{y_k}$ in Ψ. x_i are original variables in Φ and y_i are new variables that we introduce to Φ. Intuitively, y_i is the negation of corresponding x_i.

The proof that this is a valid reduction is as follows:

In the reduction above, we double the total amount of variables from $\{x_i, \ldots, x_k\}$ to $\{x_i, \ldots, x_k, y_i, \ldots, y_k\}$. In addition, we double the amount of total clauses from n clauses to $2n$. We want to show that if Ψ is exactly the same as Φ. In our configuration, if x_i is TRUE, then y_i must be FALSE as y_i is constructed as the negation of x_i. Therefore, the solution to Φ is always balanced with TRUE and FALSE. Since y_i is totally dependent on x_i, the solutions to Ψ is exactly the same as Φ.

(6 points) 2) Degree-Bounded Spanning Tree: Given a graph G and integers $(b_1, \ldots, b_n)$, find a spanning tree T for the graph such that the degree of the i^{th} vertex in the tree is at most b_i.

Proof: *We will reduce the problem ...* *to the problem ...*

Given an instance Φ *of the problem ...* *we construct an instance* Ψ *of the problem ...*

as follows ...

Terse but acceptable version:

We will reduce the problem Hamiltonian/Rudrata Path to the problem Degree-Bounded Spanning Tree (DBST). Given an instance of the problem Hamiltonian Path, G, we construct an instance of the problem DBST as follows: $G, b_i = 2, \forall i$.

Solution with more explanation:

We will reduce Hamiltonian/Rudrata Path to Degree-Bounded Spanning Tree (DBST). Given an instance of Hamiltonian Path, we have a graph $G = (V, E)$ and wish to find a path that visits every vertex exactly once (we don't need to end on the starting vertex). Construct an instance to DBST as follows: use the same graph from the Hamiltonian Path problem, G, and let $b_i = 2, \forall i$ (we can also optionally further constrain $b_1 = 1$ and/or $b_n = 1$). Then we feed this into our black-box DBST solver. If there was no solution, output no solution. If there was a solution, we get a spanning tree with degree 2 or less everywhere. But if every vertex of the spanning tree has degree 2 or less, this means the entire spanning tree must be one long path with no branches (or if it wasn't a snake-like line, one vertex would have degree of at least 3). So the output of DBST is the solution to the Hamiltonian Path problem.

The proof that this is a valid reduction is as follows:

If there is a Hamiltonian path in the graph, then this path is a spanning tree where each degree is at most 2. So it must be the case that DBST has a possible spanning tree where each degree is at most 2.

If there is a spanning tree where each degree is at most 2, then this implies there is a Hamiltonian path. Because it is spanning, it includes all vertices, and because it is a tree, it is acyclic (so this is not a cycle). If every degree is at most 2, this forces the only possible spanning tree to be a path that visits all the vertices exactly once. This is a Hamiltonian path. So if there is such a spanning tree, there is a Hamiltonian Path.

■

(6 points) 3) Disjoint Sets: Given a collection of subsets $\mathcal{S} = \{S_1, \ldots, S_m\}$ of $\{1, \ldots, n\}$ and an integer k, pick k of these subsets $\{S_{i_1}, \ldots, S_{i_k}\}$ that are pairwise disjoint, i.e., $S_{i_a} \cap S_{i_b} = \emptyset$ for all $a \neq b$.

Proof: *We will reduce the problem ... Independent Set to the problem ... Disjoint Set*

Given an instance Φ of the problem ... Independent Set we construct an instance Ψ of the problem ... Disjoint Set

as follows ...

Answer key is wrong: Have incident edges instead of incident vertices

$\Phi = G(V, E)$. Create subsets S_i for all vertices $i \in V$. The contents of the subsets will the vertex $j \quad \forall (i, j) \in E$. Subsets will contain adjacent vertices. If all elements are pairwise disjoint, then there are no edges between any no vertices and therefore no edges between all vertices. We set integer k to be the target size g of the independent set.

Proof of correctness of reduction not necessary ■