Final

- **The exam has 7 questions (you can choose one of the last 5 to skip), is worth 100 points, and will last 180 minutes.**
- We indicated how points are allocated to different parts of questions. Not all parts of a problem are weighted equally.
- Read the instructions and the questions carefully first.
- Begin each problem on a new page.
- Be precise and concise.
- The questions start with true/false, then short answer, then algorithm design. The problems may **not** necessarily follow the order of increasing difficulty.
- Good luck!

Part A

1 True/False + short justification

(2 pts per item) Are the following **true** or **false**? **Provide a brief justification (1-3 sentences) for each answer.**

(a) If the weights of all the edges in an undirected graph are increased by 1, then the shortest path from s to every vertex (also called the shortest path tree from s) remains unchanged.

False. e.g. consider when there are two paths from s to t, one using 3 edges of weight 1, and one using 1 edge of length 4. The weight of the first path is smaller, but it becomes larger when we add 1 to all weights

(b) If the weights of all the edges in an undirected graph are increased by 1, then the minimum spanning tree remains unchanged.

True. Every spanning tree uses $|V| - 1$ edges, so the ordering of spanning trees by weight doesn't change.

(c) There exists a linear program $\min c \cdot x$ subject to $Ax \geq b$ for which there is a solution with objective value 5, and for which the dual linear program has a solution with objective value 10.

False. For a minimization LP, any dual solution's objective lower bounds any primal solution's objective.

(d) If we can reduce a size-n instance of problem A to a size n^2 instance of problem B in $O(n^2)$ time (including preprocessing and postprocessing) and problem B has a linear-time algorithm, then problem A has a quadratic-time algorithm.

True. The algorithm is to run the reduction and then use the algorithm for problem B. Since the algorithm for B is linear-time and runs on a size n^2 instance, we get a $O(n^2)$ overall runtime.

For the remaining parts there are *four* possible answers:

- True,
- False
- True iff P = NP
- True iff P $\neq$ NP.

In each case, choose the right one.

(e) Consider s-t connectivity, the problem of deciding if there is a path from s to t in a graph G. There is a polynomial-time reduction from 3-SAT to s-t connectivity.

True iff P=NP. If P = NP, we can e.g. in polynomial time solve the 3-SAT, then create a graph which has a s-t path iff the 3-SAT instance is satisfiable. Furthermore, a polynomial-time reduction from 3-SAT to s-t connectivity together with the polynomial time algorithm for s-t connectivity implies a polynomial-time algorithm for 3-SAT, and thus implies P=NP.

(f) There is a polynomial-time reduction from s-t connectivity to 3-SAT.

True. Any problem in P is in NP, so it has a reduction to 3-SAT, which is NP-complete. Alternatively, one can directly state a reduction like e.g. solve connectivity and then create a trivial instance of 3SAT with the same solution.

2 Short Answer: Solve 4 out of the 5 subparts

(5 pts per item) Clearly identify which questions you solved. In case no such identification is given, we will check the first 4 questions.

(a) You run the multiplicative weights algorithm on n experts. Recall that you are guaranteed to achieve low regret comparable to the best expert. Suppose you have n competitors, and you happen to find out that on day i, competitor k follows the advice of expert $(i+k) \mod n$. Are you also guaranteed to achieve low regret comparable to your best competitor? If yes, explain why. If no, describe how to modify your algorithm to achieve the same regret bound against your competitors.

It does not also have low regret against these competitors. To have low regret against these n competitors, just run the multiplicative weights algorithm with the n competitors as experts.

(b) Consider the following algorithm for set cover: While there is some uncovered element, repeatedly choose an element x that hasn't been covered yet, and add all sets containing x to the set cover. Suppose that your set system has the property that no element appears in more than d sets. Show that the algorithm described above is a d-approximation algorithm.

Suppose the algorithm chooses k elements – then the size of the set cover it picks is at most kd. But notice that no two of the k elements it chooses appear in the same set, since otherwise when the first of these elements is picked, the other element would already have been covered. This means that at least k sets are necessary to even cover these k elements. So OPT is at least k, and so the algorithm is a d-approximation algorithm.

(Note that this is just like the maximal matching approximation algorithm for vertex cover, but generalized to set cover)

(c) Fix a prime m, and recall that $[m] := \{0, 1, 2, \ldots, m-1\}$. Let $h_{a_1,a_2} : [m] \times [m] \to [m]$ be the function $a_1 x_1^2 + a_2 x_2 \mod m$. Is the hash function family $\{h_{a_1,a_2} : a_1, a_2 \in [m]\}$ universal? If yes, briefly justify why. If no, give an example of two inputs $(x_1, x_2) \neq (y_1, y_2)$ such that $h(x_1, x_2) = h(y_1, y_2)$ with probability $> 1/m$.

It is not universal. For $m > 2$, the inputs $(1, 0)$ and $(m-1, 0)$ are distinct but always collide since $1^2 \equiv (m-1)^2 \mod m$.

(d) Suppose we have an algorithm A using $O(\log n)$ bits of memory that streams a n-element list L and outputs a random number k with expected value $\mathbf{E}[k]$ equal to the number of distinct elements in L.

Describe an algorithm using $O(\log n)$ bits of memory that streams an n-element list L_1, then streams an n-element list L_2, and outputs a random number m with expected value $\mathbf{E}[m]$ equal to the number of distinct elements that appear in both L_1 and L_2, i.e., $|\{x : x \in L_1 \text{ and } x \in L_2\}|$.

Example: $L_1 = 5, 3, 5, 1, 1, 3$ has 3 distinct elements, $L_2 = 1, 1, 2, 2, 3, 1, 4, 2, 6$ has 5 distinct elements, while there are only 2 distinct elements, namely $1, 3$ that appear in both L_1 and L_2.

Hint: For any two sets S, T, $|S| + |T| = |S \cap T| + |S \cup T|$.

We run three copies of algorithm A, call them A_1, A_2, A_3. We stream all elements in L_1 to A_1 and A_3, and all elements in L_2 to A_2 and A_3. Let k_1, k_2, k_3 be the outputs of these algorithms, we output $k_1 + k_2 - k_3$. By linearity of expectation

$$\mathbf{E}[k_1 + k_2 - k_3] = \mathbf{E}[k_1] + \mathbf{E}[k_2] - \mathbf{E}[k_3] = |L_1| + |L_2| - |L_1 \cup L_2| = |L_1 \cap L_2|.$$

The space complexity is $O(\log n)$ since we use $O(1)$ copies of A on streams of size $O(n)$.

(e) You want to multiply the two polynomials $f(x) = x$ and $g(x) = x^3$ using FFT. However, you forgot to pad appropriately, and have used vectors of length 4. What would be the resulting product of $f(x) \cdot g(x)$ according to your algorithm? Briefly justify your answer.

The FFT of the polynomial $f(x) = x$ is $[1, i, -1, -i]$. The FFT of the polynomial $g(x) = x^3$ is $[1, -i, -1, i]$. Their pointwise product is $[1, 1, 1, 1]$ and the inverse FFT on this vector gives the vector $[1, 0, 0, 0]$ which correspond to the polynomial $h(x) = 1$.

Note that one can solve this problem without doing any matrix-vector multiplications by recalling that FFT just evaluates the polynomials at roots of unity, multiplies the results, and then finds the polynomial matching these results. e.g. without doing a matrix-vector multiplication we can see x evaluates to $[1, i, -1, -i]$ at 4th roots of unity, x^3 evaluates to $[1, -i, -1, i]$ at 4th roots of unity, and the only polynomial of degree at most 3 that evaluates to $[1, 1, 1, 1]$ at roots of unity is $h(x) = 1$.

Part B : Solve 4 out of the next 5 questions

(17 pts per question) Clearly identify which questions you solved. In case no such identification is given, we will check the first 4 questions.

3 Road Trip Redux

You want to drive from San Francisco to New York City on I-80. Your car holds C gallons of gas, but is a gas guzzler and gets 1 mile per gallon. There are n gas stations along I-80, with gas station i at distance d_i miles from San Francisco (in sorted order), with cost per gallon c_i. You start with a full tank at station 1 in San Francisco, and your goal is to reach gas station n in NYC, while spending as little as possible on gas. You may assume there is some feasible trip, i.e., that $\max_i[d_i - d_{i-1}] \leq C$.

(a) Suppose the ith gas station has the cheapest price (i.e. the smallest value of c_j) among stations $2, \ldots, n-1$. Show that if we spend as little money as possible, then either the ith gas station is the first one we buy gas at or we arrive at the ith gas station with an empty tank. Similarly, argue that either the ith gas station is the last one we buy gas at or we leave the ith gas station with a full tank. You may assume the c_i values are distinct. (Hint: use an exchange argument)

Suppose by contradiction that the last time we bought gas before arriving at station i was at station i' and we arrived at the ith gas station with some gas remaining. Instead, we could have bought k less gallons of gas at station i' for some $k > 0$, and then bought k more gallons of gas at station i, overall reducing the cost of our trip while maintaining its feasibility.

(In particular, k can be the minimum of the gallons we buy at i' and the amount we arrive at i with.)

Similarly, if the next time we buy gas after leaving station i was at station i', and we left station i without filling our tank, then for some $k > 0$ we could have bought k more gallons of gas at station i and k less gallons of gas at station i', again reducing our cost while preserving feasibility of the trip.

(b) Using part a, give an $O(n^2)$-time divide-and-conquer algorithm to compute the minimum amount of money you can spend on gas. Just the algorithm description and runtime analysis are needed.

- If $d_n - d_1 \leq C$, return 0.
- Otherwise, determine the station $i \in \{2, \ldots, n-1\}$ with the cheapest c_i. Recursively determine the minimum cost to get from station 1 to i, and station i to n (if we leave i with a full tank).
- If getting from station 1 to i costs 0, we know we arrive at station i with $C - (d_i - d_1)$ gallons of gas, otherwise we arrive with 0 by part (a).
- Similarly, if getting from station i to n costs 0, this means we don't need to leave station i with more than $d_n - d_i$ gallons of gas, otherwise we should leave with C by part (a).
- Our overall cost is the cost to get from 1 to i, plus the cost to get from i to n, plus the cost of gas purchased at station i. The first two are computed by the recursive call, and the last is c_i times the difference in the number of gallons we should leave and arrive with.

It takes $O(n)$ time to identify the station with the smallest c_i, and we recurse on problems of size i and $n - i + 1$. So our runtime is given by the recurrence relation $T(n) = T(i) + T(n - i + 1) + O(n) = O(n^2)$ in the worst case.

4 LP+Dijkstra alternative to Bellman-Ford

Let $G = (V, E)$ be a directed weighted graph with possibly negative edge weights w_{ij} for edge (i, j). Our goal is to efficiently compute a new set of non-negative weights w'_{ij} such that the shortest paths between any s and t remains unchanged under the change of weights — this means that we can use Dijkstra under weights w' to compute the shortest path from s to t.

Specifically, for each vertex i we increase the weight of all outgoing edges by some amount c_i and decrease the weight of all incoming edges by c_i. This means that $w'_{ij} = w_{ij} + c_i - c_j$

(a) Argue that the shortest paths from s to t under w' are the same as the shortest paths from s to t under w. Note that the path lengths may not be the same under w and w'.

At any internal vertex v in the path from s to t, the weight of the incoming edge into v is decreased by c_v and the weight of the outgoing edge is increased by c_v, for a net change of 0. The only exceptions are s and t: at s there is no incoming edge, so there is only an increase by c_s, and at t there is no outgoing edge, so there is only a decrease by c_t. The net change is $c_s - c_t$. Since this is a constant for all paths from s to t, it does not affect which paths have the shortest length, so the shortest paths from s to t are the same under w and w'.

Alternately, you can just explicitly write out the expression for the new length:
Consider some path $(s, v_1, \ldots, v_k, t)$ from s to t. The new length of this path is given by

$$w'_{sv_1} + w'_{v_1v_2} + \cdots + w'_{v_{k-1}v_k} + w'_{v_kt} =$$

$$(w_{sv_1} + c_s - c_{v_1}) + (w_{v_1v_2} + c_{v_1} - c_{v_2}) + \cdots + (w_{v_{k-1}v_k} + c_{v_{k-1}} - c_{v_k}) + (w'_{v_kt} + c_{v_k} - c_t) =$$

$$(w_{sv_1} + w_{v_1v_2} + \cdots + w_{v_{k-1}v_k} + w_{v_kt}) + (c_s - c_t)$$

Therefore, the length of any path from s to t is merely shifted by a constant $c_s - c_t$, which doesn't depend on the path. Shifting the length of every path from s to t by a constant doesn't affect which paths have the shortest length, so the shortest paths from s to t are the same under w and w'.

(b) We know that the problem of finding shortest paths in a graph is only well-defined if the graph has no negative weight cycles. Show that if there is some choice of $\{c_i\}_{i \in V}$ such that all w'_{ij} are non-negative, then G has no negative weight cycles.

Notice that the weight of any cycle remains unchanged under the modified weights — this follows from part (a), by letting $s = t$. Therefore, if all edges have nonnegative weight in the modified graph, then all cycles do as well, so all cycles must have nonnegative weight in the original graph.

* Note that it is also true that if G has no negative weight cycles then there is some choice of $\{c_i\}_{i \in V}$ such that all w'_{ij} are non-negative. But we are **not** asking you to prove it for this question.

Add a dummy vertex v_0 and connect v_0 to all vertices $i \in V$ with an edge of weight 0. Take c_i to be the distances from v_0 to i, for $i \in V$. These distances are well-defined since G has no negative weight cycles. The distances are also finite since all vertices are reachable from v_0. For any edge (i, j) we have $c_j \leq c_i + w_{i,j}$ since a plausible path from v_0 to j is the shortest path from v_0 to i plus the edge (i, j). By rearranging this equation we see that $w_{i,j} + c_i - c_j \geq 0$ for all edges $(i, j) \in E$.

(c) We want to choose the amounts, c_i, so the adjusted weight w'_{ij} of every edge in the modified graph is non-negative. Suppose we also want to modify the weights as little as possible, i.e., we want to minimize the sum of the absolute values of the c_i's. Write this as a linear program.

$$\min_{c_i, z_i} \sum_{i=1}^{n} z_i$$

$$-z_i \leq c_i \leq z_i \qquad \forall i \in V$$

$$w_{ij} + c_i - c_j \geq 0 \qquad \forall (i, j) \in E$$

5 Small Vertex Cover

Recall that $S \subseteq V$ is a vertex cover for graph $G = (V, E)$, if every edge in E has at least one endpoint in S. The vertex cover problem asks on input $G = (V, E)$ and k whether G has a vertex cover of size at most k. While Vertex-Cover is NP-complete, we will give an algorithm for this problem that is efficient when k is small.

(a) Give an algorithm to decide if there's a vertex cover of size 1 or less in $O(|V| + |E|)$ time. Just the algorithm description is needed.

The vertex cover has size 0 iff there are no edges in the graph. Otherwise, select any edge $e = (u, v)$. One of u or v must be in the vertex cover, so if the vertex cover has size 1, then u or v must be the vertex cover. We can test whether u is the vertex cover in linear time by checking each edge to see if u is one of its endpoints. Similarly we test whether v is the vertex cover.

Alternate solution: If there's a vertex cover of size (at most) 1, there must be a vertex adjacent to all edges, i.e. with degree $|E|$. So we just count the degrees of all vertices and return "Yes" iff such a vertex exists.

(b) Give a recursive algorithm to decide if there's a vertex cover of size at most k in $O(2^k \cdot (|V| + |E|))$ time. Give the algorithm description, a brief justification, and runtime analysis. (Hint: to solve the problem with parameter k, solve two problems with parameter $k - 1$, and apply recursion.)

For half credit you may instead give an algorithm to decide if there's a vertex cover of size at most 2 in $O(|V| + |E|)$-time. Give the algorithm description, a brief justification, and runtime analysis.

If G has no edges, trivially there is a vertex cover of size 0.

Otherwise, fix an arbitrary edge (u, v). If a vertex cover of size k exists, either u or v must be in the vertex cover, and thus either $G - u$ (the graph G with u and all adjacent edges deleted) or $G - v$ must have a vertex cover of size $k - 1$. So our algorithm is to recursively decide if either $G - u$, $G - v$ has a vertex cover of size $k - 1$. The base case is when $k = 1$, in which case we use part (a).

We have $T(k) = 2T(k - 1) + O(|V| + |E|)$, since we can write down $G - u, G - v$ in $O(|V| + |E|)$ time each. The solution to this recurrence is $T(k) = O(2^k(|V| + |E|))$.

The partial credit solution is the same as the above solution, but with a fixed value of k (which means it can be expressed non-recursively).

Alternative solution for partial credit. Find a maximal matching $M \subseteq E$ in the graph $G = (V, E)$. If $|M| = 1$ then the vertices adjacent to M form a vertex cover of size 2, so the answer is "Yes". If $|M| \geq 3$ then any vertex cover for G is of size at least 3 and the answer is no. If $|M| = 2$ then any vertex cover to G must cover M. If the vertex cover is of size 2 it must choose for each edge exactly one vertex. So there are 4 potential candidates for a vertex cover given M, and we check each of them in linear time. Note that this solution doesn't generalize simply to the case $k > 2$.

6 Is-Different

Consider the following problem called **Is-Different**. You are given two 3-CNF (3-SAT) formulas[1] Φ_1, Φ_2 over the same set of variables $x_1, \ldots, x_n$. You want to decide if there is some assignment of values to $x_1, \ldots, x_n$ such that $\Phi_1 \neq \Phi_2$, i.e. one of the formulas evaluates to true and the other to false.

For example, for $\Phi_1 = (x_1 \vee x_2 \vee x_3)$ and $\Phi_2 = (x_1 \vee x_2 \vee \overline{x_3})$ the assignment $(0, 0, 0)$ causes Φ_1 to be false but Φ_2 to be true, so the answer would be **yes**. However, for $\Phi_1 = (x_1 \vee x_2) \wedge (\overline{x_2})$ and $\Phi_2 = (x_1) \wedge (\overline{x_2})$, we always have $\Phi_1 = \Phi_2$ so the answer is **no**.

[1] Recall that a 3-CNF formula is the AND of some number of clauses, and each clause is the OR of between 1 to 3 literals, e.g., $(x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_2} \vee x_4) \wedge (\overline{x_1})$.

(a) Show that Is-Different is in **NP**.

If the answer is yes, then by definition there is an assignment $a_1, \ldots, a_n$ such that $\Phi_1 \neq \Phi_2$. Given this assignment it is easy to evaluate (in polynomial time) the two formulas Φ_1, Φ_2 and check that one evaluates to true and the other to false.

(b) Show that Is-Different is **NP**-hard.

We reduce from 3-SAT. Given a 3-SAT instance Φ, we create an Is-Different instance where $\Phi_1 = \Phi$ and $\Phi_2 = (x_1) \wedge (\overline{x_1})$ (that is, Φ_2 is the trivial formula that always evaluates to false). If Φ_1 is satisfiable then it Is-Different than Φ_2, because it evaluates to true on the satisfying assignment, while Φ_2 evaluates to false. On the other hand if Φ_1 is not satisfiable then they both evaluate to false on any assignment. Therefore it is immediate that the 3-SAT formula is satisfiable if and only if the answer to Is-Different is yes.

(c) Φ_1 and Φ_2 are called equivalent if they have the same satisfying assignments, i.e., they have the same truth value for all assignments to the variables (in other words, if the answer to Is-Different is no). Consider now the problem of 3-CNF-minimization: Given a 3-CNF formula Φ, find an equivalent 3-CNF formula with the smallest number of clauses.

Show that if 3-CNF-minimization can be solved in polynomial time, then 3-SAT can be decided in polynomial time. [2]

The main idea is almost the same as the previous part: if Φ is unsatisfiable then there is an equivalent formula $(x_1) \wedge (\overline{x_1})$ with just 2 clauses. So just compute the minimal equivalent formula. If it has more than 2 clauses then Φ must be satisfiable. Else just check whether the equivalent formula with at most 2 clauses is satisfiable — this can be done in constant time.

7 Local 3-SAT

Consider the following variant of 3-SAT, called k-local 3-SAT. We are given a 3-SAT instance $\Phi(x_1, x_2, \ldots, x_n)$ with m clauses, with the guarantee that x_i, x_j appear in the same clause only if $|i - j| < k$.

(a) We will give a dynamic programming algorithm for this problem that is efficient when k is small – the target running time is $O(2^k nm)$.

In the j-th iteration of our algorithm we wish to restrict our attention to the variables $x_1, x_2, \ldots, x_j$. Denote by $\Phi_j(x_1, x_2, \ldots, x_j)$ the restriction of $\Phi(x_1, x_2, \ldots, x_n)$ to clauses containing only variables $x_1, x_2, \ldots, x_j$. For example, if $\Phi(x_1, x_2, \ldots, x_4) = (x_1) \wedge (x_1 \vee x_2) \wedge (x_1 \vee x_2 \vee \overline{x_3}) \wedge (x_1 \vee x_3 \vee x_4)$ then $\Phi_1(x_1) = (x_1)$ and $\Phi_3(x_1, x_2, x_3) = (x_1) \wedge (x_1 \vee x_2) \wedge (x_1 \vee x_2 \vee \overline{x_3})$.

Define the subproblem $f(j, a_1, \ldots, a_k)$ for every $j \in \{k, \ldots, n\}$, and $a_1, \ldots, a_k$ Boolean. $f(j, a_1, \ldots, a_k)$ is true if $\Phi_j(x_1, x_2, \ldots, x_j)$ has a satisfying assignment with $x_{j-k+1} = a_1, \ldots, x_j = a_k$.

Note that as base cases we can brute force compute $f(k, a_1, \ldots, a_k)$. Give a recurrence relation for solving these subproblems for $j > k$. Analyze the runtime of your recurrence relation. Also describe how to determine if the k-local 3-SAT formula is satisfiable given the solutions to all these subproblems.

The recurrence relation is:
$f(j, a_1, \ldots, a_k) = (f(j-1, True, a_1, \ldots, a_{k-1})$ OR $f(j-1, False, a_1, \ldots, a_{k-1}))$ AND the partial assignment $x_{j-k+1} = a_1, \ldots, x_j = a_k$ satisfies all clauses on variables from x_{j-k+1} to x_j.
This works because in $\Phi_j(x_1, x_2, \ldots, x_j)$ the new variable x_j occurs only in clauses containing variables from x_{j-k+1} to x_j. So we only need to check that those clauses are satisfied by the partial assignment $x_{j-k+1} = a_1, \ldots, x_j = a_k$, and that $\Phi_{j-1}(x_1, x_2, \ldots, x_{j-1})$ is satisfied with $x_{j-k+1} = a_1, \ldots, x_{j-1} = a_{k-1}$.

There are 2^k values for $a_1, \ldots, a_k$, n values for j, and we can solve the recurrence for one subproblem in $O(m)$ time. So the overall time is $O(2^k nm)$.

[2] By polynomial time, we mean polynomial in the number of clauses and variables.

Given the solutions to all these subproblems, the k-local 3-SAT instance is satisfiable iff $f(n, a_1, \ldots, a_k)$ is true for some $a_1, \ldots, a_k$.

(b) Give a polynomial-time reduction from 3-SAT to $\sqrt{n}$-local 3-SAT.

We take a 3-SAT instance with n variables, x_1 to x_n. Add $n^2 - n$ dummy variables $x_{n+1}, \ldots, x_{n^2}$ that don't appear in any clauses. Let $n' = n^2$. This reduction transforms the original 3-SAT instance into an equivalent $\sqrt{n'}$-local 3-SAT instance with n' variables.

Alternative reduction: If you are uncomfortable with variables not appearing in any clause you can use the following instead:
Given a 3-SAT instance $\Phi(x_1, \ldots, x_n)$, we add $n^2 - n$ new variables $x_{n+1}, \ldots, x_{n^2}$ and add one clause per new variable by taking $\Phi'(x_1, \ldots, x_{n^2}) = \Phi(x_1, \ldots, x_n) \wedge (x_{n+1}) \wedge (x_{n+2}) \wedge \ldots \wedge (x_{n^2})$. It is easy to see that Φ is satisfiable iff Φ' is satisfiable. Furthermore Φ' is a $\sqrt{n'}$-local 3-SAT instance on $n' = n^2$ variables.