Problem 1 (10 points).

Solve the "Hidden Surface Removal" problem (Problem 5, Chapter 5 on page 248 of the Textbook [KT]) using certain algorithm(s) introduced in our lectures.

Solution. We can reduce this hidden-surface-removal problem into the half-plane-intersection problem: for each of the given line $y = a_i \cdot x + b_i$ (given in the hidden-surface-removal problem) into a half-plane $y \geq a_i \cdot x + b_i$, $1 \leq i \leq n$. Clearly, a line is visible (in the hidden-surface-removal problem) if and only if the corresponding half-plane (in the half-plane-intersection problem) is part of the boundary of the intersection of all n half-planes.

Therefore, we can use the divide-and-conquer algorithm for half-plane-intersection problem (introduced in class) to find the intersection of these n half-planes. The intersection (a convex region) is represented as the list of half-planes that consist of the boundary of the convex region. This list hence, according to our above analysis, gives the set of all visible lines.

The running time of above algorithm is dominated by the running time of calling the divide-and-conquer algorithm for half-plane-intersection problem, which is $O(n \cdot \log n)$.

Problem 2 (15 points).

You are given n points $P = \{p_1, p_2, \cdots, p_n\}$ on 2D plane, represented as their coordinates. You are informed that the convex hull of P contains exactly 8 points in P (assume that $n \geq 8$). Design an algorithm to compute the convex hull of P in $O(n)$ time. You may assume that no three points in P are on the same line.

Solution. We first find the point in P with smallest y-coordinate, denoted as p_1^*. Clearly, p_1^* must be on the convex hull of P. Let $C = (p_1^*)$ store the list of points on the convex hull of P found so far. Let $p_0^* = p_1^* - (1,0)$ be a virtual point (not within P) be the point on the left of p_1^*. We then iteratively find all other points on the convex hull of P. In the k-th iteration, $k = 2, 3, \cdots$, for each point $p \in P \setminus \{p_0^*, p_1^*\}$, we compute the angle $\angle p p_{k-1}^* p_{k-2}^*$. We then find the point that maximizes this angle, denoted as p_k^*. Clearly, p_k^* must be on the convex hull of p as well (Proof. line $p_k^* p_{k-1}^*$ is an indicator: all other points locate at one side of this line). Therefore, we add p_k^* to C and continue to next iteration. The algorithm terminates when we find p_k^* equals to p_1^*.

Each iteration finds a new point on the convex hull and takes linear time. The total number of iterations equals to the number of points on the convex hull. Hence, the running time is $O(n \cdot s)$, where s is the number of points on the convex hull. This algorithm is *output-sensitive*. When there are constant number of points on the convex hull, as described in this problem, this algorithm runs in linear time.

Problem 3 (15 points).

You are given a non-convex polygon represented with its vertices $P = \{p_1, p_2, \cdots, p_n\}$ in couter-clockwise order. Design an algorithm to find the convex hull of P in $O(n)$ time. Prove that your algorithm is correct.

Solution. Assume that p_1 is the point in P with smallest y-coordinate; otherwise, we find such point in P, in linear time, and rotate P so that that point becomes the first point in P.

Call *Graham-Scan-Core (P)*; let C be the set of points returned (points remained in the stack when the algorithm terminates).

We now show that C represents the convex hull of P (i.e., C represents the vertices of the convex hull of P in counter-clockwise order). First, Graham-Scan-Core guarantees that any continuous three points on

the stack are "turning left", which implies that the points in C represents a convex polygon. Second, we show that the polygon represented by C contains all points in P. This is because the algorithm keeps the following invariant: after k-th iteration (i.e., the algorithm finishes processing point p_k), the points remained in the stack followed by points that have not yet been processed (i.e., $p_{k+1}, p_{k+2}, \cdots, p_n$), form a polygon (not necessarily convex) that contains all points in P. In other words, when the algorithm pops out the top element p of the stack, p must be within the polygon. Combined, C represents a convex and smallest polygon (as all points in C are from P) that contains all points in P. Therefore, by definition, C represents the convex hull of P.

Note: Although the points in P might not be sorted wrt the *angle*, we can still use it in our case. The only thing we need—which is essentially what graham-scan-core does—is to remove vertices so that any three consecutive are "turning left". With this property and that the points in P are the vertices of a polygon, we can prove that the returned points are the vertices of the convex hull of P (above paragraph).

Problem 4 (20 points).

You are given two sets of points $P = \{p_1, p_2, \cdots, p_n\}$ and $Q = \{q_1, q_2, \cdots, q_n\}$ on 2D plane. Design an algorithm to determine whether there exists a non-vertial line L such that all points in P locate in one side of L while all points in Q locate in another side of L. Your algorithm should run in $O(n \cdot \log n)$ time.

Solution. If such L exists, then either P are above L and Q are below L, or P are below L and Q are above L.

We first test the first case. Consider the dual problem: let $S = \{s_1, s_2, \cdots, s_n\}$ be the dual lines of P, and let $T = \{t_1, t_2, \cdots, t_n\}$ be be dual lines of Q. Then there exists L in the primary plane such that P are above L and Q are below L if and only if in the dual plane there exists a point x (dual of L) such that x is above all lines in S and x is below all lines in T. This is equivalent to decide whether the intersection the upper part of all lines in S and the lower part of all lines in T is empty, which can be reduced to the half-plane-intersection problem.

Specifically, for each line $y = ax + b$ in S we create half-plane $y \geq ax + b$ and for each line $y = ax + b$ in T we create half-plane $y \leq ax + b$. We then use the divide-and-conquer algorithm for half-plane-intersection problem to find the intersection of all these $2n$ half-planes. If the intersection is not empty, i.e., the list of lines represented by the algorithm is not empty, then such point x exists (in the dual plane), and consequently such L exists (in the primal plane). And otherwise such L does not exist.

The second case can be tested symmetrically.

Problem 5 (20 points).

Given an array A with n positive distinct integers, design an algorithm to count the number of pairs $(A[i], A[j])$ satisfying that $2 \cdot i \leq j$ and $A[i] \geq 2 \cdot A[j]$. Your algorithm should run in $O(n \cdot \log n)$ time.

Solution. (optional; won't be counted towards the final grades)

We scan A from left to right. Let the current index be j. Whenever j is even, we push $A[j/2]$ into an *binary search tree*, where each node additionally stores the size of its left sub-tree. Additionally, at step j we search $A[j]$ in the binary search tree to get the size of its left sub-tree (rooted at $A[j]$), which is the number of pairs corresponding to $A[j]$, i.e., the number of is satisfying $2 \cdot i \leq j$ and $A[i] \geq 2 \cdot A[j]$. Adding them together gets the answer.

In general, we perform $n/2$ insertion operations and n search operations in a BST, which leads to $O(n \cdot \log n)$ time complexity.

Problem 6 (15 points).

Let $G = (V, E)$ be a directed graph with positive edge length $l(e) > 0$ for any $e \in E$. For vertex $v \in V$ there is also an associated positive *vertex weight* $w(v) > 0$. For any path we define its *length* as the sum of the length of its all edges plus the sum of the weights of its all vertices. Given $s \in V$, design an algorithm runs in $O(|V| \cdot |E|)$ time to find the shortest paths (with the new definition of length) from s to all other vertices.

Solution. For each edge $e = (u, v)$ in G, we add the weight of node v to edge e. The weight of e would be $l'(e) = l(e) + w(v)$. Then we can run Dijkstra's algorithm on G starting from s with these updated edge length. After that for every vertex $v \in V$, its eventual distance from s needs to be augmented by $w(s)$. If we use binary heap within the Dijkstra's algorithm, the running time of the above algorithm is $O((|V| + |E|) \cdot \log |V|) = O(|V| \cdot |E|)$.

Alternatively, we can create a new graph $G' = (V', E')$. For each (weighted) vertex $v \in V$ in G, we split it into two unweighted vertices v_1 and v_2 and add them to V'. Then we add an edge (v_1, v_2) to E', which is assigned to have weight $w(v)$. For each edge $e = (u, v) \in E$ in the original graph G, we replace it with edge (u_2, v_1) in the new graph G'. Therefore in the new graph G' we has $|V'| = 2 \cdot |V|$ and $|E'| = |V| + |E|$. Now we can run an existing algorithm on G' to find the shortest paths from s_1.

Problem 7 (30 points).

Let $c_1, c_2, \cdots, c_n$ be various currencies. For any two currencies c_i and c_j, there is an exchange rate $r_{i,j}$; this means that you can purchase $r_{i,j}$ units of currency c_j in exchange for one unit of c_i. These exchange rates satisfy the condition that $r_{i,j} \cdot r_{j,i} < 1$, so that if you start with a unit of currency c_i, change it into currency c_j and then convert back to currency c_i, you end up with less than one unit of currency c_i (the difference is the cost of the transaction). Give an efficient algorithm for the following problem: given a set of exchange rates $r_{i,j}$, and two currencies s and t, find the most advantageous sequence of currency exchanges for converting currency s into currency t. (Assume that, there does not exists sequence of currencies $c_1, c_2, \cdots, c_k$ such that $r_{1,2} \cdot r_{2,3} \cdot r_{3,4} \cdots r_{k-1,k} \cdot r_{k,1} > 1$.)

Solution. We can transform the currency exchange problem into the shortest path problem. We build a directed graph $G = (V, E)$, where $V = \{c_1, c_2, \cdots, c_n\}$ represents all currencies, and E contains all pairs (c_i, c_j), $1 \le i \ne j \le n$. We assign length for edge $(c_i, c_j) \in E$ as $-\log(r_{i,j})$. We now show that computing the shortest path from s to t in G actually gives the optimal sequence of currency exchanges for converting s into t. In fact, there is one-to-one correspondence between a path from s to t in G and a sequence of currency exchange for converting s into t. Moreover, the length of such a path p equals to $\sum_{(c_i,c_j) \in p} -\log r_{i,j} = -\log \prod_{(c_i,c_j) \in p} r_{i,j}$. Hence, the shortest path in G gives the path maximizes $\prod_{(c_i,c_j) \in p} r_{i,j}$, which is exactly the maximized amount of currency t that can be coverted from a unit of currency s. Based on the above analysis, the algorithm will be to compute the shorest path from s to t in G (using any of the single-source shortest path algorithm we introduced). The vertices along this optimal path gives the sequence of currencies following which we can get the maximized amount of currency t.

Occasionally the exchange rates satisfy the following property: there is a sequence of currencies $c_1, c_2, \cdots, c_k$ such that $r_{1,2} \cdot r_{2,3} \cdot r_{3,4} \cdots r_{k-1,k} \cdot r_{k,1} > 1$. This means that by starting with a unit of currency c_1 and then successively converting it to currencies c_2, c_3, $\cdots$, c_k, and finally back to c_1, you would end up with more than one unit of currency c_1. Give an efficient algorithm for detecting the presence of such an anomaly.

Solution. We need to detect whether there is $r_{1,2} \cdot r_{2,3} \cdot r_{3,4} \cdots r_{k-1,k} \cdot r_{k,1} > 1$, which is equivalent to detect $-(\log(r_{1,2}) + \log(r_{2,3}) + \log(r_{3,4}) + \cdots + \log(r_{k-1,k}) + \log(r_{k,1})) < 0$. Since we assign length for edge $e = (c_i, c_j)$ as $-\log(r_{i,j})$, this exactly implies a negative cycle in G. In other words, such an anomaly exists

if and only if G contains negative cycles. Therefore, we can use Bellman-Ford algorithm on G to identify whether G contains negative cycles, which also answers whether there exists anamaly.

Problem 8 (20 points).

In each of a continuous period of n days, an easy task and a hard task is given. You can choose to work on the easy task, which gives you reward a_k, $1 \le k \le n$, work on the hard task, which gives you reward b_k, $1 \le k \le n$, or choose neither of them, which of course gives you reward 0. If you choose to work on a hard task on day k, $1 \le k < n$, then you need to rest on the next day, i.e., choose to work on neither of the tasks on day $(k+1)$. Given a_k and b_k, $1 \le k \le n$, design a dynamic programming algorithm to schedule the task on each day so that the total rewards is maximized. Define the subproblems, give the recursion, describe the initialization, iteration, and termination steps of the algorithm, and give the running time of your algorithm.

Solution. Define $A[k]$ as the maximum rewards can be obtained in the first k days if at day k easy task is chosen. Define $B[k]$ as the maximum rewards can be obtained in the first k days if at day k hard task is chosen. Define $C[k]$ as the maximum rewards can be obtained in the first k days if at day k none of the task is chosen. The recursion is given below:

$$\begin{aligned} A[k] &= a_k + \max\{A[k-1], C[k-1]\} \\ B[k] &= b_k + \max\{A[k-1], C[k-1]\} \\ C[k] &= \max\{A[k-1], B[k-1], C[k-1]\} \end{aligned}$$

The *pseudo-code*, which includes three steps, is as follows.

Initialization:

$A[1] = a_1$
$B[1] = b_1$
$C[1] = 0$

Iteration:

for $k = 2 \cdots n$:
  $A[k] = a_k + \max\{A[k-1], C[k-1]\}$
  $B[k] = b_k + \max\{A[k-1], C[k-1]\}$
  $C[k] = \max\{A[k-1], B[k-1], C[k-1]\}$
endfor

Termination: $\max\{A[n], B[n], C[n]\}$ gives the maximized total awards can be obtained.

Running time: $O(n)$.

Problem 9 (20 points).

A string is *palindromic* if it is the same whether read left to right or right to left. Given a string A of length n, design a dynamic programming algorithm to find the longest palindromic subsequence of A. (For example, the longest palindromic subsequence for string $A = ACTCGCATA$ is $ATCGCTA$.) Define the subproblems, give the recursion, describe the initialization, iteration, and termination steps of the algorithm, and give the running time of your algorithm. Your algorithm should run in $O(n^2)$ time.

Solution. Define $c[i, j]$ as the length of the *longest palindromic subsequences* within substring $A[i \cdots j]$. For

$i+1 \leq j$, we have the *recursion* formula below (we define $\delta(A[i] = A[j]) = 1$ if $A[i] = A[j]$ and $\delta(A[i] = A[j]) = 0$ if $A[i] \neq A[j]$):

$$c[i,j] = \max \begin{cases} c[i+1, j-1] + 2 \cdot \delta(A[i] = A[j]) \\ c[i+1, j] \\ c[i, j-1] \end{cases}$$

The *pseudo-code*, which includes three steps, is as follows.

Initialization:

$c[i, i-1] = 0$, for all $1 \leq i \leq n$
$c[i, i] = 1$, for all $1 \leq i \leq n$

Iteration:

for $d = 1 \cdots n-1$:
  for $i = 1 \cdots n-d$:
    $j = i + d$
    $c[i,j] = \max\{c[i+1, j-1] + 2 \cdot \delta(A[i] = A[j]), c[i+1, j], c[i, j-1]\}$
  endfor
endfor

Termination: $c[1,n]$ gives length of the *longest palindromic subsequences* in A. (The corresponding indices can be fetched through introducing backtracing pointers.)

Running time: The initialization of $c[i,j]$ takes $O(n)$ time. The iteration step takes $O(n^2)$ time. The total running time of the algorithm is $O(n^2)$.