# Problem Set 10

**All parts are due on December 6, 2018 at 11PM**. Please write your solutions in the LaTeX and Python templates provided. Aim for concise solutions; convoluted and obtuse descriptions might receive low marks, even when they are correct. Solutions should be submitted on the course website, and any code should be submitted for automated checking on `alg.mit.edu`.

---

**Problem 10-1.** [22 points] **Merging In Place**

Merge sort efficiently sorts an array of comparable items by repeatedly calling a linear-time merge operation. Merging uses an external array that is the same size as the original array, so merge sort is **not in-place**[1]. In this problem, you will use merge sort and merging to sort an array in-place. Define an array to be $(a, b)$**-sorted** if the the left-most $a$ items appear in sorted order and the right-most $b$ items also appear in sorted order. For example, array $A = [5, 6, 8, 2, 1, 7, 3, 4]$ is $(a, b)$-sorted for every $0 \le a \le 3$, $0 \le b \le 2$.

**(a)** [8 points] Given an array $A$ containing $n$ comparable items, describe how to **adapt merge sort** to permute $A$ in-place[1] into a permutation $A'$ that is $(k, 0)$-sorted, for any integer $k \le n/2$. Your algorithm should run in $O(k \log k)$ time. For simplicity, throughout this problem, you may assume that $n$ is a power of two.

**(b)** [2 points] Show how to use part (a) twice to permute $A$ in-place[1] into a permutation $A'$ that is $(n/4, n/2)$-sorted.

**(c)** [8 points] Given an array $A$ containing $n$ comparable items, where $A$ is $(a, n - 2a)$-sorted for some integer $a \le n/4$, describe an $O(n)$-time algorithm to permute $A$ in-place[1] into a permutation $A'$ that is $(0, n - a)$-sorted.

**(d)** [4 points] Show how to (repeatedly) use parts (a), (b), and (c) to sort an array $A$ containing $n$ comparable items in-place[1] in $O(n \log n)$ time. Is your algorithm stable?

[1] Recall that an in-place algorithm uses no more than constant additional space beyond its input, i.e. array $A$.

**Problem 10-2.** [10 points] **Catching Culprits**

Retail giant Azamon has figured out that many MIT students create WebMoira mailing lists to sign up for arbitrarily many free Azamon Deluxe student accounts. Azamon is interested in identifying sets of email addresses that correspond to the same user. For every purchase made, Azamon logs three fields: an email address, a credit card number, and an IP address associated with the transaction[2]. If any of these fields is shared between two different purchases, Azamon will assume they were made by the same user. Given $p$ Azamon purchase logs, describe an $O(p)$-time algorithm to return a list of user records associated with the purchases, where each user record is a list of all email addresses associated with one user. State whether your running time is worst-case, amortized, and/or expected.

**Problem 10-3.** [12 points] **Mega Crush Training**

Amos Saran wants to improve her gameplay in the crossover fighting game, **Mega Crush Sisters**, by training against players that have similar abilities. Each player in the Mega Crush Sisters ladder (including Amos) has a unique integer ID as well as an integer ***winningness***: the number of games the player has won minus the number of games they have lost. Note that two players may have the same winningness. Amos is only willing to train with players whose winningness differs from her own by at most $\pm 10$, and she prefers to train with a player who has recently played in a tournament (to ensure they are active in tournament play). Design a database to help Amos keep track of the $n$ players supporting the following two operations, each in $O(\log n)$ time. State whether the running time of each operation is worst-case, amortized, and/or expected.

1. `change_wins(i,k,t)`: record that player with ID $i$ scored $k$ tournament wins at time $t$ (or losses if $k$ is negative).
2. `find_partner(i)`: return the ID of a player who played in a tournament **most recently**, from among all players having winningness within $\pm 10$ of the player with ID $i$.

**Problem 10-4.** [12 points] **Rapid Rounding**

Grick Rimes is the sheriff of the capital colony of Running Life, a healthy country comprising $n$ colonies connected by $r$ roads, where each road connects one pair of colonies, each road is at most $n^2$ miles long, and each colony is connected to at most $8$ roads. The capital's governor asks Grick to produce a list of the minimum distances a citizen would need to run on roads from the capital to each colony in Running Life. Grick runs around the country so much that, by instinct, he already knows an **estimate** of the shortest running distance to each colony from the capital, equal to the exact distance rounded to the nearest $10$ miles. Grick notices that no more than $5$ colonies have the same rounded estimate. Given a map of Running Life marked with the exact length of each road and Grick's rounded distance estimate for each colony, describe an $O(n)$-time algorithm to compute exact distances to each colony from the capital. State whether your running time is worst-case, amortized, and/or expected.

[2] You may assume that each purchase log, and thus each field, is storable in a constant number of machine words.

**Problem 10-5.** [24 points] **Dynamic Programs for APSP**

This problem considers three dynamic-programming approaches to solve the All-Pairs Shortest Paths (APSP) problem: given a weighted directed graph $G = (V, E, w)$, with possibly negative weights but **no negative cycles**, compute $\delta(u, v)$ for all pairs of vertices $u, v \in V$. Assume vertices are identified by consecutive integers such that $V = \{1, 2, \ldots, |V|\}$.

**(a)** [8 points] An approach similar to Bellman–Ford uses subproblems $x(u, v, d)$: **the smallest weight of a path from vertex $u$ to $v$ having at most $d$ edges**. Describe a dynamic-programming algorithm on these subproblems to solve APSP in $O(|V|^4)$ time.

**(b)** [8 points] Consider subproblems $y(u, v, i) = x(u, v, 2^i)$: **the smallest weight of a path from vertex $u$ to $v$ having at most $2^i$ edges**. Describe a dynamic-programming algorithm on these subproblems to solve APSP in $O(|V|^3 \log |V|)$ time.

**(c)** [8 points] Consider subproblems $z(u, v, k)$: **the smallest weight of a path from vertex $u$ to $v$ which only uses vertices from $\{1, 2, \ldots, k\} \cup \{u, v\}$**. Describe a dynamic-programming algorithm on these subproblems to solve APSP in $O(|V|^3)$ time.[3]

**Problem 10-6.** [20 points] **Longest Alternating Subsequence**

Integer sequence $(x_0, x_1, \ldots, x_{n-1})$ is **alternating** if its elements satisfy either:

$$x_0 > x_1 < x_2 > x_3 < \cdots x_{n-1} \qquad \text{or} \qquad x_0 < x_1 > x_2 < x_3 > \cdots x_{n-1}.$$

**(a)** [8 points] Given integer array $A = (a_0, a_1, \ldots, a_{n-1})$, describe an $O(n^2)$-time dynamic-programming algorithm to find the longest alternating subsequence[4] of $A$.

**(b)** [10 points] Describe a data structure to store items, each containing a key and a value, supporting the following two operations, each in $O(\log n)$ time, where $n$ is the number of items stored at the time of the operation.

1. `insert(x)`: add item $x$ to the data structure (where $x$ contains a key and a value).
2. `max_value_under(k)`: return an item with largest value, from among all stored items having key strictly less than $k$.

**(c)** [2 points] Show how to use the data structure from part (b) to speed up your algorithm from part (a) to run in $O(n \log n)$ time.

[3] This algorithm is known as **Floyd-Warshall**. Do not reference external presentations of this algorithm when answering this question; please follow our dynamic-programming framework instead!

[4] This problem is similar to **Longest Increasing Subsequence** from Recitation 16. You may read through those notes for reference, but your solutions to this problem should be complete on their own, without citing those notes.