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 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.

Solution: Merge sort merges two sorted subarrays by transferring their contents into an auxiliary array (an external array of the same size), and then transfers the sorted contents back to the original array. Instead of using external auxiliary space, we alternatively use the second k elements of A as the auxiliary space for merge sorting the first k elements. Instead of overwriting the items in the auxiliary space, the algorithm swaps all items from $A[k : 2k]$ with all items from $A[: k]$ during the merge, and then swaps them back. Since merge sort takes $O(k \log k)$ time to sort $O(k)$ items, so does this algorithm, without using more than a constant amount of additional space outside of the original input array A.

Rubric:

- 6 points for a correct algorithm
- Full proof of correctness not required
- 2 points for a correct running time analysis
- Partial credit may be awarded

(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.

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

Solution: Use part (a) to produce a $(n/2, 0)$-sorted array from A. Then swap the first $n/2$ items with the last $n/2$ items by swapping item i with item $n/2 + i$ for $1 \leq i \leq n/2$, so the new array is $O(0, n/2)$-sorted. Finally, use part (a) again to sort the first $n/4$ items, without touching the last $n/2$ items, resulting in an array that is $O(n/4, n/2)$-sorted.

Rubric:

- 2 points for a correct reduction
- Partial credit may be awarded

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

Solution: We use the merge step of merge sort to merge subarrays $A[: a]$ and $A[n - 2a :]$ into subarray $A[n - a :]$, swapping elements during the merge. The merge step of merge sort maintains the invariant that after the kth swap, the k-smallest elements from $A[: a]$ and $A[n - 2a :]$ exist sorted in subarray $A[a : a + k]$. The merge step of merge sort takes $O(n)$ time, thus so does this algorithm.

Rubric:

- 6 points for a correct algorithm
- Full proof of correctness not required
- 2 points for a correct running time analysis
- Partial credit may be awarded

(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?

Solution: Use part (b) to permute A into a $(n/4, n/2)$-sorted array B in $O(n \log n)$ time (which is $(k, n - 2k)$-sorted array for $k = n/4$). Then use part (c) to permute B into array C which is $(0, n - k)$-sorted for $k = n/4$ in $O(n)$ time. Then use part (a) to permute C into an array which is $(k, n - 2k)$-sorted for $k = n/8$ in $O(k \log k)$ time. Repeat parts (c) and (a) in this way, each time dividing k in half until $k = 1$, and the array is $(1, n - 2)$-sorted. Lastly, swap the first two elements up into their sorted location in $O(n)$ time. The running time for this algorithm is then $O(n \log n) + T(n/4) + O(n)$, where $T(k) = T(k/2) + O(n) + O(k \log k)$. Then since $O(n) + O(k \log k)$ is upper bounded by $O(n \log n)$, $T(n/4)$ is also upper bounded by $T'(n)$, where $T'(k) = T'(k/2) + \Theta(n \log n)$. Then by case 3 of the Master Theorem where $a = 1$, $b = 2$, and $f(n) = \Theta(n \log n)$, $T'(n) = \Theta(n \log n)$, so this algorithm runs in $O(n \log n)$ time. (This running time could also be computed directly via a summation). Using other parts of the array as auxiliary space for merging may arbitrarily permute items originally in the auxiliary space (even items having equal keys), so this algorithm is not stable.

Rubric:

- 2 points for a correct reduction
- 1 points for a correct running time analysis
- 1 points for correct argument for unstable
- Partial credit may be awarded

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.

Solution: We build a graph G on unique fields such that each user record will be a connected component. Add each field in the purchase logs to a hash table, mapping each unique field in expected $O(p)$ time to an associated graph vertex and a Boolean specifying whether the field is an email address. Then for each purchase log containing three fields (i, j, k), add undirected edges $\{i, j\}$, $\{j, k\}$, and $\{k, i\}$ to G (if they do not already exist). Then email addresses i and j are in the same connected component of G if they correspond to the same user. Run either breadth-first search or depth-first search to fully explore each connected component of G in linear time. While searching each connected component, construct a new user record as a dynamic array, adding the email address corresponding to every black vertex in the connected component. Then return these records. Graph G has at most $3p$ vertices and at most $3p$ edges, so this algorithm takes expected $O(p)$ time.

Note that using radix sort to find duplicates does not work for this problem. Word size must be at least $\log n$, but it is not restricted to be $O(\log n)$, so you cannot assume fields are polynomially bounded in n. Maximum 8 points should be assigned for a correct solution claiming use of a linear-time sorting algorithm.

Rubric:

- 8 points for a correct algorithm
- 2 points for a correct running time analysis
- (0 running time points for a radix-based approach)
- Partial credit may be awarded

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

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 ± 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 ± 10 of the player with ID i.

Solution: Record each player as a triple containing ID, current winningness, and most recent tournament play time. Store each player in an AVL tree of max heaps, where each max heap stores players having the same winningness ordered by recent tournament time, and the AVL tree sorts max heaps by their associated winningness. In addition, maintain a hash table from each player ID to where that player is stored in the tree. The AVL tree stores at most n winningness values, and each max heap contains at most n players, while the total data structure has size $O(n)$, since each player exists in at most one max heap. To implement `change_wins(i,k,t)`, lookup in the hash table the location of player i in the data structure in expected $O(1)$ time, and then remove that player from the max heap in which it is stored. Then add k to its winningness, and replace recent tournament play time with t if t is more recent. Then reinsert the player into the data structure by finding player's new winningness in the AVL tree, and then inserting the player into its max heap, each in $O(\log n)$ time. Lastly, updating the hash table with the player's new location in the data structure results in an expected $O(\log n)$-time operation. To implement `find_partner(i)`, search in the AVL tree for the node corresponding to the winningness w of player i, and then repeatedly run successor and/or predecessor to find the at most $21 = O(1)$ AVL nodes corresponding to being within ± 10 of w, all of which can be found in $O(\log n)$ time. Then, look at the players stored at the roots of the max heaps associated with these nodes, to return any player having most recent tournament play in $O(1)$ time, resulting in a worst-case $O(\log n)$-time operation.

Alternate solutions: One could use an AVL tree in place of a hash table for the ID linking data structure, which would lead to worst-case $O(\log n)$ performance for both operations. The top level of the nested data structure could be a hash table instead of an AVL tree since you just need to check $21 = O(1)$ values, leading to an expected time bound for `find_partner`. Both of these replacements are acceptable as long as the student argues correctly for either expected or worst-case. Additionally, inner data structures can be AVL trees instead of max heaps achieving the same bounds. Alternatively, one could use a single AVL tree storing players sorted by winningness supporting two-sided range queries, where each node v is augmented with the node x in v's subtree corresponding the player having played in a tournament most recently. This augmentation can be maintained in $O(\log n)$ time and can be used to find a partner in $O(\log n)$. This solution would still require a dictionary mapping IDs to nodes in this augmented AVL tree, so that a player can be found efficiently.

Rubric:

- 4 points for a correct data structure
- 3 points for a correct algorithm for `change_wins`
- 1 point for a correct running time analysis for `change_wins`
- 3 points for a correct algorithm for `find_partner`
- 1 point for a correct running time analysis for `find_partner`
- Partial credit may be awarded

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.

Solution: Create graph G having colonies as vertices and edges corresponding to roads between the colonies weighted by their exact length. Since each colony is connected to at most a constant number of roads, G has size $O(n)$. We could simply run Dijkstra on this graph to compute the shortest paths in $O(n \log n)$ time, but we are asked for $O(n)$ time. Observe that all the edge weights are no more than n^2, so all shortest paths from the capitol are no more than n^3. Thus, we can sort the rounded distance estimates using radix sort to find an ordering of colonies in (approximately) increasing distance from the capital in $O(n)$ time. Then run a procedure similar to topological sort relaxation using this ordering, iterating through each colony and relaxing all outgoing edges. If multiple colonies have exactly the same rounded distances, then the exact distance ordering of these vertices is not known. However, since there are at most 5 colonies with the same rounded distance, we can relax all outgoing edges from colonies having the same estimate in 4 rounds (similar to Bellman-Ford), ensuring that afterward, all edges from those colonies are then fully relaxed. This algorithm relaxes every edge at most 4 times, so this algorithm takes in worst-case $O(n)$ time (assuming we store our graph in a way that allows the adjacency list of vertex i to be found in worst-case $O(1)$ time, for example by labeling each colony with an integer from 1 to n and storing adjacency lists in a direct access array).

Instead of using multiple rounds like Bellman-Ford, one could also simply run Dijkstra locally on a set of equal-rounded vertices, relaxing edges from each in order of smallest shortest path estimates computed so far; since there are only at most 5 vertices, the vertex to be processed next can be found in constant time, leading to the same running time as above.

Rubric:

- 8 points for a correct algorithm
- 2 points for a brief argument of correctness
- 2 points for a correct running time analysis
- Partial credit may be awarded

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.

Solution:

2. **Relate:** $x(u, v, d) = \min\{x(u, k, d-1) + w(k, v) \mid (k, v) \in E\} \cup \{x(u, v, d-1)\}$ (only depends on smaller d so acyclic)
3. **Base:** $x(u, u, 0) = 0$, $x(u, v, 0) = \infty$ for $u \neq v$
4. **Solution:** $x(u, v, |V| - 1)$ for all $u, v \in V$
5. **Time:** $O(|V|^3)$ subproblems, each $O(|V|)$ work: $O(|V|^4)$ in total

Rubric:

- 3 points for a correct recurrence
- 1 point for acyclic (only if recurrence correct)
- 2 points for correct base cases (only if subproblems correct)
- 1 point for solution (only if subproblems correct)
- 1 point for running time (only if recurrence correct)
- Partial credit may be awarded

(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.

Solution:

2. **Relate:** $x(u, v, i) = \min\{x(u, k, i-1) + x(k, v, i-1) \mid k \in |V|\} \cup \{x(u, v, i-1)\}$ (only depends on smaller i so acyclic)
3. **Base:** $x(u, u, 0) = 0$, $x(u, v, 0) = w(u, v)$ if $(u, v) \in E$, or ∞ otherwise
4. **Solution:** $x(u, v, \lg |V| + 1)$ for all $u, v \in V$
5. **Time:** $O(|V|^2 \log |V|)$ subproblems, each $O(|V|)$ work: $O(|V|^3 \log |V|)$ in total

Rubric:

- 3 points for a correct recurrence
- 1 point for acyclic (only if recurrence correct)
- 2 points for correct base cases (only if subproblems correct)
- 1 point for solution (only if subproblems correct)

- 1 point for running time (only if recurrence correct)
- Partial credit may be awarded

(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]

Solution:

2. **Relate:** $x(u, v, k) = \min\{x(u, k, k-1) + x(k, v, k-1), x(u, v, k-1)\}$ (only depends on smaller k so acyclic)
3. **Base:** $x(u, u, 0) = 0$, $x(u, v, 0) = w(u, v)$ if $(u, v) \in E$, or ∞ otherwise
4. **Solution:** $x(u, v, |V|)$ for all $u, v \in V$
5. **Time:** $O(|V|^3)$ subproblems, each $O(1)$ work: $O(|V|^3)$ in total

Rubric:

- 3 points for a correct recurrence
- 1 point for acyclic (only if recurrence correct)
- 2 points for correct base cases (only if subproblems correct)
- 1 point for solution (only if subproblems correct)
- 1 point for running time (only if recurrence correct)
- Partial credit may be awarded

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.

Solution:

1. **Subproblems**
   - Choose prefixes as subproblems
   - $x(i, s)$: length of the longest alternating subsequence B from the first i integers of A, where the last two integers of B are a_j and a_i, and $sa_j < sa_i$, for $s \in \{+1, -1\}$.
2. **Relate**

[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.

- To solve $x(i, s)$, the second to last integer in a longest subsequence ending in a_i must be a_j for some $1 \leq j < i$ satisfying $sa_j < sa_i$, so guess!
- $x(i, s) = \max\{x(j, -s) \mid 1 \leq j < i \text{ and } sa_j < sa_i\} + 1$
- Subproblems $x(i, s)$ only depend on strictly smaller i, so acyclic

3. **Base**
   - $x(i, s) = 1$ if $sa_j \geq sa_i$ for all $1 \leq j < i$
4. **Solution**
   - Solve subproblems via recursive top down or iterative bottom up
   - Solution to original problem is $\max\{x(i, s) \mid 1 \leq i \leq n, s \in \{+1, -1\}\}$
   - Store parent pointers to reconstruct a maximizing subsequence
5. **Time**
   - # subproblems: $O(n)$
   - work per subproblem: $O(n)$
   - time to compute solution is $O(n)$
   - $O(n^2)$ running time

Rubric:

- 2 points for correct subproblems
- 2 points for a correct recurrence
- 1 point for acyclic (only if recurrence correct)
- 1 points for correct base cases (only if subproblems correct)
- 1 point for solution (only if subproblems correct)
- 1 point for running time (only if recurrence correct)
- Partial credit may be awarded

(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.

Solution: Store each item in an AVL tree sorted by key, where each node v stores an item $v.x$ and is augmented with $v.a$, any item having largest value of items within v's subtree. This augmentation is maintainable because it can be computed in constant time from the augmentation on its children (when they exist):

$$v.a = \arg\max_{y \in \{v.x, v.right.a, v.left.a\}} (y.value) \ .$$

To insert an item, simply insert it into the AVL tree, maintaining the augmentation during the update in worst-case $O(\log n)$ time. To query `max_value_under(k)`, perform a one-sided range query for item with largest value having key less than k. Specifically, start at the root maintaining an item y, initially None, and recursively repeat the following operation. If the item at the current node v has key strictly less than k, then the item we are looking for is either y, $v.x$, the item with largest key in v's left subtree, or exists in v's right subtree; so set y to $\max\{y, v.x, v.left.a\}$, and recurse on $v.right$. Alternatively if $v.x$ has key k or larger, then y must exist in v's left subtree (if such an item exists); so recurse on $v.left$. If no child exists on which to recruse, then return y. This process walks down the tree, so it takes at most worst-case $O(\log n)$ time.

Rubric:

- 2 points for a correct data structure description
- 3 points for a correct algorithm for `insert`
- 1 point for a correct running time analysis for `insert`
- 3 points for a correct algorithm for `max_value_under`
- 1 point for a correct running time analysis for `max_value_under`
- Partial credit may be awarded

(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.

Solution: We can use two of the data structures from part (b) to reduce the work per subproblem in the relate step of part (a) from $O(n)$ time to $O(\log n)$ time. The first data structure D_{+1} stores items having key a_i and value $x(i, -1)$, while data structure D_{-1} stores items having key $-a_i$ and value $x(i, +1)$. Compute subproblems $x(i, +1)$ and $x(i, -1)$ bottom up from 1 to n; whenever a subproblem is computed, add it to its respective data structure. Then, we can compute the recursive step for $x(i, s)$ by calling `max_value_under(`a_i`)` on D_s in worst-case $O(\log n)$ time. By doing the computation bottom-up, processing index i ensures that each D_s only stores $x(j, s)$ for $j < i$, so calling `max_value_under` finds the maximum value $x(j, -s)$ for all $j < i$ with key sa_j such that $sa_j < sa_i$, leading to $O(n \log n)$ total running time.

Rubric:

- 2 points for using part (b) correctly to optimize recurrence from (a)
- Partial credit may be awarded