Introduction to Algorithms: CS 1205
Ashoka University
Debayan Gupta Neeraj Pandey

Neeraj Pandey

Name: Neeraj Pandey

Problem 1-1.

(a) **False**.
Reason: This is incorrect second case of the Masters theorem. The recurrence relation will be of T = $O(\sqrt(n) * log^2 * n)$

(b) **True**
Reason: $O(logn)$ is less than $O(n^10)$ asymptotically, and the n here means the size here $(\geq 1). Therefore an algorithm with the time complexity of$ O(n^{1}0) will take time at least $O(logn)$.

(c) **True**
Reason: This is true because searching in heaps takes the time complexity of $O(logn)$ and heaping in worst case takes $O(n)$. Therefore, the time complexity will be $O(\frac{n}{\log n} \times \log n) = O(n)$.

(d) **True**
Reason: This is because if the higest node is the top node and the second highest is the left child to it and this continues. Then, the max element will be on the top of the max heap.

(e) **False**
Reason: This is because the hash tables in the average or expected time is in the time complexity of $O(1)$.

(f) **False**
Reason: Time complexity of AVL tree is $OElogV + VlogV)$ when used with Dijkstra algorithm, but when used with fibonacci heap the time complexity is $O(E + VlogV)$.

(g) **False**
Reason: DFS can't find the shortest path in an unweighted graph. BFS can be used to find it instead.

(h) **True**
Reason: Yes, we can find the single source shortest paths by using Topological sort.

(i) **True**
Reason: Yes, because quick sort is a divide and conquer algorithm and not Dynamic Programming.

(j) **True**
Reason:

Problem 1-2.

(a) With the given array of integers with elemetns from 1 to the largest element as n^m, we can perform radix sort or merge sort.
The given time complexity is $O(n \times min\{m, \log n\})$.
This could be simplified as

$$O(n) * O(min(m, logn))$$

$$= O(m * n) * O(n * logn)$$

Also, time complexity of merge sort is $O(nlogn)$, time complexity of radix sort is $O(nm)$, and the time complexity of comparing the two (radix and merge) will take $O(1)$. Therefore, if $m < logn$, we can use radix sort for the given array, otherwise we should use a merge sort to sort the given array and the total time complexity will be equal to $O(n * min(logn, m))$.The correctness of the algorithm can be referenced from the Radix Sort of the class notes.

(b) To solve this we will create a min-heap data structure. The algorithm is shown below:

- First, initialise the min-heap with $k + 1$ elements and an empty array/list.
- Remove the minimum element from the min-heap and insert it into an array.
- Once the element is removed, heapify the min-heap (replace that node with the next node and continue..), and repeat the process.
- Continue repeating it unless there is no node left in the min-heap and all the nodes are moved to the array. The values in the array will be sorted eventually by doing the above steps.

While running this algorithm, construction of min-heap will take $O(k)$ time, heapify will take $O(logk)$ times, removing of node from the min-heap will take $O(1)$. This entire will be done n times. So, the overall time complexity will be equal to $O(nlogk)$. Correctness for the algorithm can be referenced from the Min-Heap part from the class notes.

Problem 1-3. Here, we will construct a graph $G = (V, E)$. The vertices here are the bakers and edges are the paths between two bakers in the community and the graph is a complete graph because the Monster can travel to all node to and fro. Given that the Brute Monster's sleepiness decreased by 10minutes. The time taken to travel from one baker to the other baker is given as m_{ij}. Therefore, each edge weight can be considered as the sleepiness amount which will be equal to $m_{ij} - 10k$. Here the sleepiness of the monster depends on two factors that it will increase by 1minute every minute he travels from one baker to the other and decreases by 10 min each time he eats a biscuit. The edge weight (sleepiness amount) here could be negative as well in a scenario when the Monster eats more cookies and very less distance travelled. Here, we have to minimize sleepiness, we have to minimize the path travelled by the monster to travel from one baker to another while eating more biscuits. So, considering all factors, we can use Bellman Ford Algorithm to find the shortest path to all the vertices. To perform the algorithm, we take all the vertices with the sleepiness amount and relax each pair starting from the source taken the vertex of baker b_1. Here, relaxation increases the accuracy of the distance of the given pair of vertices. It works by continuously shortening the calculated distance between the vertices that distance with other known distances. So, we will run Bellman ford algorithm from the source node as b_1 with intial edge weight as -1 and find the minimum distance to b_n. The time complexity for this algorithm will be $O(VE)$ for V vertices and E edges, and as the graph is a complete graph, $O(E)$ = $O(V^2)$. Therefore final time complexity will be $O(V^3)$.

Problem 1-4. The basic idea behind the algorithm is to run DFS from each vertex to find the paths to the other vertices and in DFS, we traverse each node exactly once. So, this will take $O(V)$. If there is blue doesn't not exist, we create a blue edge to that vertex. Detailed explanation: So, we create a copy of the given graph and add all the blue edges so that it becomes a blue-friendly graph and includes all the original nodes in it. Doing this will exclude some of the red edges from the original graph, so we will be running DFS from each vertex on this graph and the graph will return all the connected components in that graph. Now, every pair of vertices that are connected to each other and has a blue edge, then all the nodes in both the vertices components are blue friendly. By doing this, there might be some some pairs that were not connected by red edges in the original graph, We will find such pairs and connect them. To perform this algorithm, the time complexity will be equal to $O(V(V-1)/2)$ for the maximum edges in the undirected graph with V vertices and we run DFS on the graph in $O(V+E)$ and checking the color of the E edges in $O(E$ = $O(V(V-1)/2)$. Total time complexity will be $O(V+E)*O(E)$ = $O(V^2)$

Problem 1-5. We can solve this problem using Dynamic programming:
Subproblem: DP[i][j][k] = minimum difficulty sum from node i to node j such that the sum of the length % 60 == k

SOlution: Solving subproblems via recursive top down or iterative bottoms up
For bottom up, we can solve it in the increasing order of f adn then e,
So, the final solution will be DP[p-junction][g-junction][0]
Time complexity: Time complexity to perform this will be equal to $60 * n * n$ = $60n^2$

Problem 1-6.

God bless me!

Problem 1-7.

Subproblem:

- •We have to find two teams of size $s/2$ and combat power of $m/2$ where m is given as $p_1......p_s$ and check if there exists a team subset of sum as $m/2$ and length as $s/2$.
- •Now, define an 2-D array DP for the dimensions $\big((m/2)+1\big) * \big(s+1\big)$

Relate:
DP[i, j] = True, if DP[i-1][j-y] is true where y belongs to the combat strength from 1 to s
else, DP[i,j] = False where j ¡ i
Base Case:
Here we will use prefix sum, so, if it divides equally then DP[0][0] = 0 and True
otherwise, DP[0][i] will be False i can be range from 0 to $m/2$

Solution:
The subproblems will use iterative bottom up approach which will be solved in order of increasing f, after which it will be in increasing e.
So, the final solution wil be DP[$\frac{m}{2}$, s].
If there is a team of 2 equal size it will be equal to 0.

Time complexity:
Time complexity for the sub problem will be $O(s * m)$ and for every subproblem we traverse the arrays once, so it will take time $O(s)$. Therefore, total time complexity will be equal to $O(s^2 * m)$.

Problem 1-8. This problem can be solved using hashing and by creating a hash table. The steps to implement the algorithm:

- Create a hash table with each pair of unique minions names.
- We will now check for buddy names. So, if the hash values of a buddy pair is same then there exist buddy buddies which means four minions with the same buddy name, it means that their buddy name or the concatenated name will be identical.
- Now, iterate through k keys for a maximum length of name k in the hash table to check the length of the list with values that are equal to 4.
- That way, we will find a group of buddy buddies and max combination of names (pairs) could be $n * n = n^2$.
- So, time complexity to perform this algorithm will be $O(k)$ for hashing of k buddy names and $O(n^2)$ for constructing the hash table for all the buddy combination pairs. Total time complexity will be equal to $O(kn^2)$.

Problem 1-9. The basic idea behind the algorithm is to basically keep track of the last access time stored with a variable that can use a hashmap. Also, maintaining a balanced search tree of access times. Whenever an element comes, we remove it's last occurrence and check what was the rank of the last occurrence (which is the number of elements before it). Then add new access and check rank, and just report the different. This will use Hashmap and AVL Trees / Treemap. Hash map is considered as $O(1)$ and accessing from an AVL tree is of time complexity $O(logn)$. We can also use tree map instead of AVL tree, that will also take $O(logn)$. So, the total time complexity will be equal to $O(logn)$. To go into a little more detail, we have to run a loop from i = 0 ; i ¡ length(value) - 1 and then check if a key exists or not, if it exists then we can save the current value in a local temporary variable and add it to the list and return one less than the difference of the new index value and the old index value. If the key doesn't exists, then add the key with the value of index of the loop and return. All this can be added to the hash table and AVL trees will store the details of the last access and current accesss.