Data Structure Design Questions

1 Kartikland: The Happiest Place in the World

You are a ticket ripper at the happiest place in the world, Kartikland. Each family in your land can rip exactly one ticket, no more no less. Additionally, you want to rip the tickets of younger people before older ones, after all, Kartikland is a place for young ins to drag their parents to spend lots of money. Families may attempt to get around this by sending family members with tickets, but it is up to you, the brave ticket ripper to prevent them from getting more than one ripped ticket. You are provided the familial connections of all families in whatever representation you would like. You should be able to rip the tickets of all people in O(N) time where N is the number of people in line to get their tickets ripped.

1.1 Solution

Use a modified Weighted Quick Union With Path Compression to store family relationships. Create a wrapper class for each person in line and store a pointer to their corresponding Weighted Quick Union with Path Compression. When reaching a person in line, check their family Weighted Quick Union With Path Compression. If the root is marked do not rip their ticket, otherwise rip the ticket and then mark the root.

2 Sweet Tooth

Pretend that you are the owner of a Sweet Company. Your company makes three sweets- popcorn, candy, and ice cream. In addition to this, each Sweet Category has their own sub categories. For example, you can have Vanilla Ice Cream, Chocolate Ice Cream, Banana Ice Cream etc. Each Sweet has a total of N sub categories. You know how much money each sub category will yield- assume that the money each sub category will yield is unique.

As an owner trying to make a profit, you want to get lots of money; however, you can only make one item a day, and you only have M days (where M< N). Your strategy is to pick the sub category of food that will provide the most money on and then create it that day. You have 2 restrictions though.

- On any 2 consecutive days, you cannot pick items that come from the same Sweet Category. For example, you cannot create an item from Ice Cream one day and the next day pick another item from ice cream. This is enforced until the other 2 Sweet Categories are depleted.
- Once you make a subcategory, you cannot make it again. For example, if you make Banana Ice Cream one day, you can never again make Banana Ice Cream.

Utilize Data Structures that we have discussed earlier to create an algorithm that will allow you to pick the most profit while following the above constraints. You must take no longer than Mlog(N) to pick all the sub categories of Sweets you will make. Hint: You cannot simply put everything inside 1 Priority Queue.

2.1 Solution

We will build a maxheap for each category, where the root of each heap will be the maximum profit sub category.

1. On the first day we perform a removeLargest operation on the maxheap with the largest root. We keep track of the maxheap that this item is removed from
2. We then perform a removeLargest operation on the heap with a larger root out of the 2 heaps that were not picked in the prior step.
3. We continue step 2 until we reach M days.

3 Min-nie Mouse and Max-xie Mouse

Use or modify a data structure or set of data structures to do the following operations in the provided time:

- insert: O(log(N))
- getSmallest: O(1)
- getLargest: O(1)
- deleteSmallest: O(log(N))
- deleteLargest: O(log(N))

3.1 Solution

We will fulfill all of our requirements by using a basic Balanced Search Tree. By construction, Balanced Search Trees insert in O(log(N)) time and can also delete in O(log(N)) time. As a result, the only operations that we need to account for is getSmallest and getLargest. We can trivially keep a pointer that keeps track of the smallest element and the largest element. Whenever we delete the smallest element or update the Balanced Search Tree to have a smallest element, we will update our pointer. We can do a similar process for the largest element.

The reason why 2-3 trees are not used in the place of normal min-heaps or max-heap are really easy to implement, as they can be visualized using arrays. This also saves a lot of memory compared to Balanced Search Trees which can tend to be a bit expensive.

4 Oh Hi Mark

Imagine you are implementing a basic social media. You want to be able to check if 2 people are friends in constant time. You also want to be able to see if you can see if 2 people have at least 1 mutual friend in O(N) time where N is the total amount of people on the social media. Find an efficient solution to this problem.
(A slightly more clear explanation)

4.1 Solution

To solve this problem, we will associate each user with a weighted quick union with path compression. To check if 2 people are friends, we would go to one of the users and do a simple find operation. This would take almost amortized O(1) time. To see if people have a mutual friend, we would need to go through each one of the first person's friends and see if they are friends with the second person. We would perform find in each one of the first person's friends. This upperbounds the work by O(N).

An interesting way to solve this problem is a more algorithmic approach. You can compute a unique prime number for each person on your social media upon construction. To find N prime numbers takes Nlog(log(N)) time. This would be faster than the weighted quick union with path compression implementation as we would construct a data structure for each item, which would take $O(N^2)$ time.

In addition to the associated prime number, each person would have a "friend product" attribute which would be the product of all of it's friends' primes. To check if two people are friends we can just divide the friend product of one person by the prime number associated with the other. If it the result is an integer, we know that they are friends, if not, they are not friends. To see if two people have mutual friends, we will find the greatest common divisor using Euclid's Algorithm. This will take $O(log(min(a, b))$ where a is the friend product of the first person and b is the friend product of the second person.

5 Colise-Yum

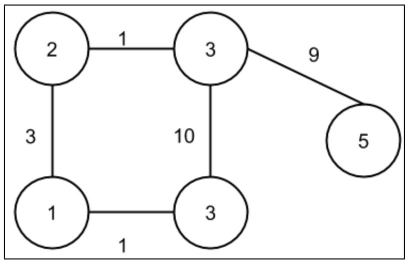
You are the rich Roman Emperor Jewelius Caesar. You want to build arenas (possibly more than one) so your subjects can watch gladiator duels and eat wild boars.

- You have V locations where you can build arenas and it costs v_i denarius's to build an arena in the ith location.
- Additionally, any location that does not have an arena must have a path to an arena. You are provided a set of roads, denoted by E, that you can build between any 2 locations i and j, along with their cost denoted by e_{ij}.

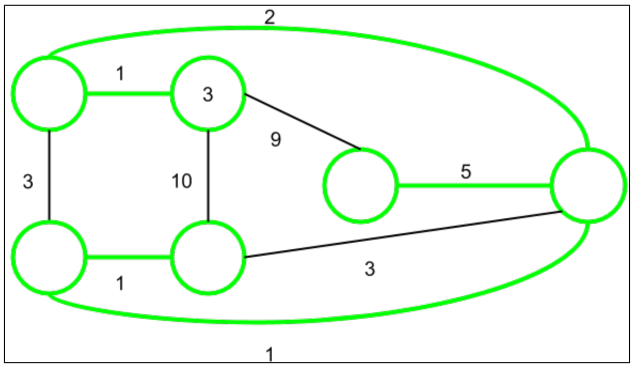
Find or modify an algorithm that will minimize the amount of denarius's your kingdom will spend while still fulfilling your constraints.

5.1 Solution

Essentially, this can be thought of as a graph problem where we want to "mark" nodes and have nodes either be connected to a marked node or be marked themselves, whichever is cheaper. To do this, we create a graph with one vertex per location with edges between them corresponding to the cost of building a road between them. The graph with vertex weights would look as follows:



To account for the vertex weights, we will add one dummy node and connect it to every single location. The weight of the edge dummy node to any vertex v_i would be the cost to build in the location v_i. Now we simply run Kruskal's or Prim's and we will get a valid solution. Below is a picture of the original graph followed by a picture of the graph with the included dummy node.



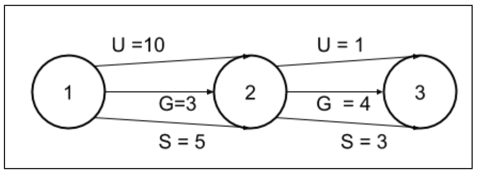
6 Land Lubber? Sea Lubber? Air Lubber? Just Lubber

You are an avid traveler who wants to travel between the 2 cities in your country which consists of V cities. To go between these cities you have decided that you can narrow down your modes of transportation to dinghy sailing, gliding, and unicycling. For every city in the country, you are given a list of cities that you can get to- this list remains constant regardless of which mode of transportation you pick. The set of all sailing costs is labeled as E_s, the set of all gliding costs is given ans E_g, and the cost of all unicycling costs is the set E_u.

Sadly, you get sea/air/landsick quite frequently so you cannot take the same mode of transportation consecutively- in other words you cannot go into city i with unicycling and leave unicycling. Find an algorithm that computes the fastest route from city i to city j with your constraints.

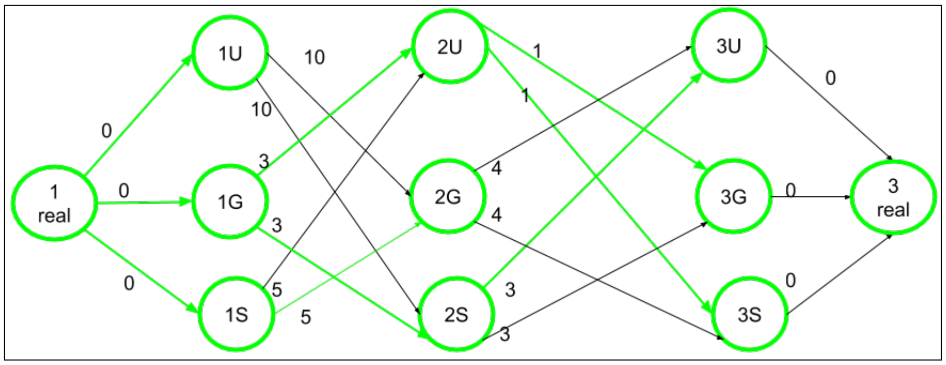
6.1 Solution

Say our original graph is G. Make three copies of the graph, one for each mode of transportation- let's label the graph for sailing G_s with vertices denoted as v_s, the graph for unicycling as G_u with vertices v_u, and the graph for gliding as G_g with vertices v_g. The cost to travel between 2 cities can be thought of as a directed edge between these 2 cities. For example, in the below, we would like to find the route we should take from 1 to 3 (On the edges, U stands for the unicycle cost, G for the gliding cost, and S for the sailing cost) .



- For all edges (u,v) in E_g create an edge between u_g and v_s as well as an edge between u_g and v_u. The weight of this edge should be the cost to glide from u to v.
- For all edges (u,v) in E_s create an edge between u_s and v_g as well as an edge between u_s and v_u. The weight of this edge should be the cost to sail from u to v.
- For all edges (u,v) in E_u create an edge between u_u and v_g as well as an edge between u_u and v_s. The weight of this edge should be the cost to unicycle from u to v.

Create a dummy start node, i_{real} that connects to i_g, i_s, and i_u with a weight of 0. Additionally create a dummy end node, j_{real} that connects to j_g, j_s, and j_u. We run dijkstra's starting at i_{real} and get the distance to j_{real}. The reason that this algorithm works is that we only connect one vertex to another if any edge coming out of it will not have the same mode of transportation. Additionally, the shortest path will be returned because we are just running Dijkstra's on this expanded graph. Below is how the new graph would look along with the solution.



7 National Bongoola Association

You are the coach of a prestigious Bongoola team. Your job as coach is to make sure that, at any given time, the best players on your team for each positions are on the field. Bongoola, being a great community team sport, invites all people of the community to join. Each player will be assigned a position that they can play- there are a total of M positions- and a unique team number. For this problem, we will use N to be the total amount of people on your team and P to be the people signed up for a certain position.

You have your starting M players, but as the game progresses, you will need to replace them. You will base your replacement of people using the bodacious factor. During the game, only 1 player's bodacious factor goes down, and once they are benched, their bodacious factor doesn't change. Every Z minutes, 1 player on the field will have their bodacious factor decrease by some unknown amount and you will be able to replace the player if needed. Once the next eligible player's (a player who plays the same position) bodacious factor exceeds the current player, they are considered a better choice. You are guaranteed that only 1 of the players at a time will need to be replaced. Finding all the players on your current lineup should take $\Theta(M)$ time, changing a player's bodacious factor should take $\Theta(1)$ time, replacing a player should take $\Theta(log(P))$ time, and putting a new player in the current lineup should take $\Theta(1)$ time.

Write a detailed description of what data structures you will use to make the operations stated above run in the provided time. State any assumptions you need to regarding anything (no you cannot assume that everything is done by some magical warlock).

7.1 Solution

First thing is first, you will need a Player class with a bodacious instance variable. You will then create a HashMap which will hash based off the unique player id's. Because of the HashMap, we can change the bodacious factor of a player in $\Theta(1)$ time. For this part of the problem, you need to state that you assume that there is a good hashcode and good spread- if this was not the case, it would be possible to have $\Theta(N)$ runtime.

To represent the players who can play in a certain position, we will use a priority queue with the priority equal to players' bodacious factors. The top player in the priority queue will be in the current lineup. Once their bodacious factor is less that one of their children, we will sink the player. This results in $\Theta(log(P))$ time to replace a player (if they truly need to be replaced).

Now it gets a little tricky. Since there are M positions, we cannot disregard the amount of positions as a constant. To get around this, there are three possible (intuitive) solutions:

The first solution involves realizing that there are as many priority queues as there are positions. You can create 2 M sized arrays- one array will be the current lineup and the other array will be composed of priority queues in the same order. You would modify the Player class so that it has an instance variable that stores the index that it would go to when replacing or being replaced. Accessing an array would occur a maximum of 2 times which would make the runtime $\Theta(1)$ time.

The second solution keeps track of the lineup with an M sized array and players have an instance variable that points to their priority queue. Instead of having a separate array for all the priority queues, players have an instance variable equal to the index in the array that they belong to. When a player needs to be replaced, the new maximum item in the priority queue would be put in their index of the current lineup array.

The third solution would be slightly more complicated. You can create 2 HashMaps, again, one of the current lineup and one of the priority queues. However, since the hashing of priority queues is not straightforward, we would create a wrapper class for priority queues and create some instance variable that was unique to each priority queue. For example, we could have a string that would be the name of the position it represents. We then hash the M wrapper classes. Accessing a priority queue would take $\Theta(1)$ time if we assume a good hashcode. If good hashcode was not assumed, then it would take O(M) time

The difficulty in this solution (specifically when you attempt to hash the lineup) is that players who replace others would need to have the same hashcode. To do this you could make the key in this HashMap the position of a player and the value is the specific player. Change the equals method in the current lineup hashmap so if 2 players have the same key they are considered equal. This means when you add a new player in, they will automatically replace the player in the corresponding position. Replacing the item in the HashMap would take $\Theta(1)$ time since we know where it hashes to.

8 Future-Roaming

You are a croissant delivery boy in the 31st century. You want to deliver your croissants as quickly as possible; however, because you lost your client's addresses and they live all across the galaxy, so you don't want to go door to door. You have the phone numbers of all the people who ordered croissants from you, so you will attempt to use this to figure out in where you should go. Each phone number is about 10000000000000000000000 digits long and you have a total of N clients. You will use the following information about telephone numbers to help:

1. The sector code (the leftmost 10 digits) will allow you to figure out which sector a client is in. On average, there are a total of $\sqrt{N}$ households you need to deliver to in the sectors.
2. In the 31st century, telephone numbers have a cool feature where the difference between the last 4 digits of any 2 telephone numbers in the same sector is equal to the distance between their two corresponding households.
3. You know the address of 1 of the houses you need to go in each sector.

You want to go to sectors based off how many of your clients are in them and how close it is . If a sector has 5 clients and is 2 Megadistances away, it is more appealing than a sector that has 1 client and is 1 Megadistance away. If a sector has 6 clients and is 2 Megadistances away, it is less appealing than a sector with 9 clients and 3 Megadistances. If a sector has 4 clients and it is 2 Megadistances away, it is considered equal to a sector that has 2 clients that is 1 MegaDistance away, at this point, you just pick which one to go to randomly . You will deliver to all the households in that sector before going to the next sector. You also know how many megadistances are between each sector, and you can assume that you will start at a location that is equidistant from all the sectors.

Provide an implementation where figuring out which households are in which sector in $\Theta(N)$ time, figuring out the distance between a house and all the other houses in a sector, must take, on average, $O(N)$ time, and delivering croissants to all the households on average, $O(N^{\frac{3}{2}} \log N)$ time.

Assume that the time it takes you to walk 1 Minidistance (inside of a sector) is the same amount of time it takes your spaceship to travel one Megadistance (between sectors). Justify why your design works in the given time.

8.1 Solution

What you first want to do is run a special version of MSD on all the telephone numbers. This version of MSD would stop sorting after we get 10 iterations, since we would have sorted by the 10 most significant digits of each telephone number, and have all the numbers sorted by sector. We are using a radix sort and we are not doing it all the way through.

Additionally, since we are performing only a partial radix sort, our work done will be O(3*N) which is just O(N), fitting our constraint for figuring out which houses are in which sector.

We will denote the amount of households in a sector by the letter T.

For each sector, we will create a graph with a total of T vertices. Following this, for each household in the sector, we will find the absolute value of the difference between it and every other household in the sector.

After finding the absolute value of the difference between the 2 vertices, we would connect these two households with an edge of that weight. In (somewhat detailed) pseudocode, the above looks like:

```
for(int i = 0; i < number of households; i++){
      for(int k= i + 1; k < number of households; k++){
         int distance = absolute value(last 4 digits of i's telephone # last 4 digits of k's
            addEdge of weight distance between i and k;
         }
}
```

On average, there are N households per sector. This means that for each household in the sector, on average, we perform N comparisons to create edges between it and another household. Our recurrence relation would look as follows:

$$\sum_{i=1}^{\sqrt{N}} \sqrt{N} - i = \sqrt{N} + \sqrt{N} - 1 + \sqrt{N} - 2 ... \sqrt{N} - \sqrt{N} + 1 = O((\sqrt{N})^2) = O(N)$$

We will denote the amount of sectors with the letter M

Following this, we will create a graph with all the M sectors; however, the "weight" of each edge would be equal to the Megadistance to the sector / amount of deliveries to make in that sector, essentially reducing this problem to "which path maximizes the amount of Megadistance/amount of deliveries". This means that the more deliveries we have in a sector, the smaller the value, which means in Dijkstra's it would be prioritized more.

We would perform Dijkstra's from the sector on the graph of sectors and at we would perform Dijkstra's at each sector that we stop in. In the average case, there are a total of N sectors and N households per sector. The Dijkstra's between the sectors will take $O(Nlog(N))$ and each sector, running Dijkstra's will take $O(Nlog(N))$ time. Since, we know that there are a total of N sectors, we can multiply our sector Dijkstra runtime by N getting $O(N * Nlog(N)) = O(Nlog(N))$. We can add the terms to get the following:

$$O(Nlog(\sqrt{N}) + \sqrt{N}log(\sqrt{N})) = O(Nlog(\sqrt{N})) = O(Nlog(N^{\frac{1}{2}}) = O(Nlog(N^{\frac{1}{2}})) = \frac{1}{2}Nlog(N)) = O(Nlog(N))$$

Easy Mistakes to Make (less or no credit given for these solutions)

1. Attempting to perform full MSD or any form of LSD would be incorrect. Our digits are simply too large to perform either one efficiently. The reason why a truncated version of MSD works is because we fit the problem so that only the first digits were considered (essentially making it a constant factor).

2. Constructing a Trie would not work for seeing which numbers contain the same 10 digit prefix because we would have to go all the way down the Trie to see the last few digits that we need.

3. Not figuring out that the weight of an edge should be changed from just the distance or adding up/subtracting the amount of people in a sector and the distance. When adding the 2 values, it is impossible to know if you're dealing with a greater distance or amount of deliveries. When subtracting you could end up with negative edges/cycles. The ratio that we created allows us to properly scale the impact that the amount of people in each sector has on the algorithm.