CS2040S: Data Structures and Algorithms

Discussion Group Problems for Week 9

For: March 16–March 20

Problem 1.
(Priority queue.)

There are situations where, given a data set, we want to know the top k highest value elements. A possible solution is to store all n elements first, sort the data set in $O(n \log n)$, then report the right-most k elements. Give an algorithm to: (i) find the top k largest elements better than $O(n \log n)$; (ii) find the top k largest elements as the elements are streaming in (for each new element that is given to you, your data structure must be ready to answer queries for the top k largest elements efficiently), and the algorithm runs faster than $O(n \log n)$.

Solution: For part (i), we can quick-select the k^{th} largest element in expected $O(n)$ time. The elements that are required will be found between this element and the end of the array.
For part (ii), we can maintain a min priority queue of no more than k elements. For every element that is given to us, add it into the priority queue. If the priority queue contains more than k elements, keep removing the smallest element until the priority queue size is k. Both insert and remove-min operations run in $O(\log k)$ time since the priority queue contains at most k elements. The overall complexity is $O(n \log k)$. Certain languages (such as C++) use this idea to implement `partial_sort`.

Problem 2.
(Horseplay.)
(Relevant Kattis Problem: https://open.kattis.com/problems/bank)

There are m bales of hay that need to be bucked and n horses to buck them.

The i^{th} bale of hay has a weight of k_i kilograms.

The i^{th} horse has a strength s_i—the maximum weight, in kilograms, it can buck—and can be hired for d_i dollars.

Bucking hay is a very physically demanding task, so to avoid the risk of injury every horse can buck at most one bale of hay.

Determine, in $O(n \log n + m \log m)$, the minimum amount, in dollars, needed to buck all bales of hay. If it is not possible to buck them all, determine that as well.

Solution: Let's sort the bales of hay by non-increasing weight, and similarly sort the horses by non-increasing strength. Assume henceforth that $k_1 \geq k_2 \geq \cdots \geq k_m$ and $s_1 \geq s_2 \geq \cdots \geq s_n$.

Consider the first bale of hay. Which horses can buck this bale of hay? They are the horses $1, 2, \ldots, i-1$, where i is the first smallest index j such that $k_1 > s_j$.

Among all these horses, we should choose the cheapest one and remove it from consideration.

Now, consider the next bale of hay. Which horses can buck this bale of hay? Again, they are the horses $1, 2, \ldots, i'-1$ where i' is the first index j such that $k_2 > s_j$. Note that this is a superset of the previous set of horses—the i' here is not less than the previous i. (Crucially, any horse that can buck the i^{th} bale of hay can also buck the j^{th} bale of hay for all $j > i$.) Among all these horses, we should choose the cheapest one that hasn't previously been chosen.

We simply perform the same process until we have bucked all bales of hay. To do this efficiently, we should maintain a (min-heap) priority queue p. Every time we consider a new bale of hay, we should insert to p the costs of the horses which can now be considered, then pop the one with the minimum cost. (If the priority queue is empty then the task is of course impossible.)

It is easy to prove that this algorithm is optimal with an exchange argument. It should also be easy to see that this runs in $O(n \log n + m \log m)$ assuming s_i, k_i and d_i are fixed-width integers.

Problem 3.
(DFS/BFS.)

Recall that when performing DFS or BFS, we may keep track of a `parent` pointer that indicates the very first time that a node was visited. Explain why these edges form a tree (i.e., why there are no cycles).

Solution: A total of $n-1$ new nodes will be discovered starting from the source node. With each discovery, the `parent` pointer can be thought of as an edge. The final structure will consist of n nodes and $n-1$ edges and will form one connected component. The resultant tree is sometimes known as a BFS/DFS-spanning tree.

Problem 4.
(Graph components.)
(Relevant Kattis Problem: https://open.kattis.com/problems/countingstars)

Given an undirected graph $G = (V, E)$ as an adjacency list, give an algorithm to: (i) determine if the graph is connected; (ii) return the number of connected components (CC) in the graph.

Solution: For part (i), if the graph is connected, it means that we can pick any node and run BFS/DFS from it, and every other node must be reachable from that starting node.
For part (ii), for every unvisited node, run BFS/DFS from that node and flag every visited node as visited. All of the visited nodes form one connected component. Search for the next unvisited node and repeat the process. If there is exactly one connected component, the graph is said to be connected.

Problem 5.
(Good students, bad students.)
(Relevant Kattis problem: https://open.kattis.com/problems/amanda)

There are good students and bad students[1]. And at the end of every year, we have to divide the students into two piles: G, the good students who will get an A, and B, the bad students who will get an F. (We only give two grades in this class.)

To help with this process, your friendly tutors have each created a set of notecards, each containing two names of students (from the entire class). They promise that for every notecard, one of the two names is a good student, and the other is a bad student. Unfortunately, they do not indicate which is which. Also, since the notecards come from thirty eight different tutors, it is not immediately certain that the cards are consistent. Maybe one tutor things that Humperdink is a good student, while another tutor thinks that Humperdink is a bad student. (And Humperdink may appear on several different cards.) In addition, they do not provide cards for every pair of students; they only provide a selection of pairs. Assume you can read the names on a card in $O(1)$ time.

Devise an algorithm for determining: (i) Are the notecards are consistent, i.e., is there *any* way that we can assign students to G and B that is consistent with the cards? (ii) Are the notecards sufficient, i.e., is there one or more than one way of assigning students to the sets G and B? (iii) Assuming there are more good students than bad students, and assuming that the notecards are consistent and sufficient, determine which set each student belongs in.

Solution: This problem is about determining whether a graph is bipartite. One simple way of solving this problem is as follows: choose any student, and begin performing a BFS; for each node, the first time it is visited in the BFS, label it with its level (i.e., the root is level 0, its neighbors are level 1, its neighbors neighbors are level 2, etc.). Then scan the graph: If any node with an even level has a neighbor with an even level, then the data is inconsistent. If any node was not visited during the BFS, then the data is not sufficient. Otherwise, check whether there are more students at even levels or odd levels. If, for example, there are more students at even levels than odd levels, then assign students at even levels to G and at odd levels to B. If you find the same number of students at both even and odd levels, report an error. (This can all be done in one BFS, rather than multiple scans, but it is often easier to think about and explain as a multi-pass algorithm.)

[1] No, not really. This sort of binary distinction is silly.

Problem 6.
(Word games.)
(Relevant Kattis problem: https://open.kattis.com/problems/sendmoremoney)

Consider the following two puzzles:

- Puzzle 1:

```
    S E N D
 +  M O R E
 ----------
  M O N E Y
```

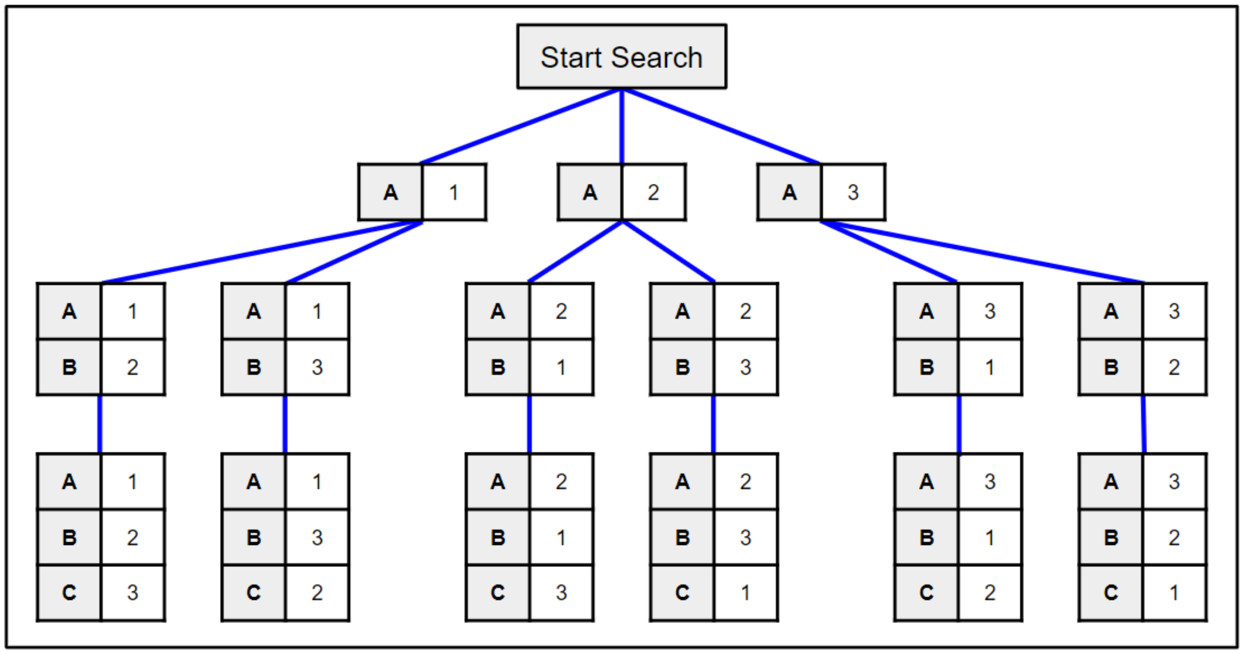
- Puzzle 2:

```
    F O R T Y
        T E N
 +      T E N
 ------------
    S I X T Y
```

In each of these two puzzles, you can assign a digit (i.e., $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$) to each of the letters in the puzzle such that the given equation holds. (Each digit is only assigned to once letter.) The goal is to solve these puzzles. How should you model and solve these puzzles? What is the running time of your solution? Can you optimize your solution to find the answer more quickly, most of the time?

Problem 6.a. Explain how to model the problem as a graph search problem. What are the nodes? How many nodes are there? What are the edges? Where do you start? What are you looking for?

Solution: There are many solutions, but the standard idea here is that each node in the graph contains some subset of the letters and some assignment of digits to letters. We might consider nodes that have increasing subsets of letters. For example, if the letters in the equation are A, B, C and the available digits are $\{1, 2, 3\}$, we might consider the following nodes:



Notice that here I have assumed that each digit can only be used once. The edges connect nodes where you can reach one from the other by assigning one more digit. For example, the node $(A, 1)$ has two outgoing edges to $(AB, 12)$ and $(AB, 13)$. You also have a distinguished start node (that is empty) that connects to all the nodes where only one letter is fixed.
You may notice that this assignment actually yields a tree! That simplifies matters. If you instead represent each node as containing every possible subset of letters and every possible assignment, then you will get a graph (i.e., both $(A, 1)$ and $(B, 2)$ will be connected to $(AB, 12)$).
For each of the nodes where all the letters are fixed, you can determine whether it is a *valid* or an *invalid* node by determining if the equation holds.

Problem 6.b. To solve this problem, should you use BFS or DFS? Why? How else can you make it run faster?

Solution: DFS is a better search algorithm, though there is no asymptotic difference. Especially for the graph representation above, BFS is equivalent to searching the entire graph. In that case, you might as well simply examine all the assignments. Using DFS, the idea will be to not search the entire graph.
In practice, for these types of exponential sized problems, you want to use various heuristics to decide which branches to explore first, and you want to rapidly trim bad directions.
For example, if you ever assign enough letters to check part of the equation, you should do so right away, and then stop exploring that path if it fails! (This is the obvious "pruning" strategy.) And you should direct your search to go visit nodes faster that you can prune. For example, if you have already assigned the letters R and T, the next letter you should assign is X so that you can verify whether or not the addition works (and prune it if not).

Problem 6.c. When does your search finish? Can you optimize the algorithm to minimize the amount of searching?

Solution: For every node, you can examine the partial assignment and decide whether it is already invalid—no possible further assignment can successfully solve the equation. For every column if the equation, if all the letters are assigned numbers, you can sum up the column (taking into account the potential carry) and determine whether it is possibly correct. If not, then abort the DFS, and do not continue exploring the sub-graph.

Problem 7.
(Graph modeling.)
(Relevant Kattis Problem: https://nus.kattis.com/sessions/qxixc8/problems/nus.cs3233mh)

Here are a bunch of problems. How would you model them as a graph? (Do not worry about solving the actual problem. Just think about how you would model it as a graph problem.) Invent some of your own problems that can be modeled as graph problems—the stranger, the better.

Problem 7.a. Imagine you have a population in which some few people are infected with this weird virus. For any two patients, you want to decide whether the infection might have spread from one to the other via some intermediate asymptomatic patients. You suspect that if such a path exists it isn't too long (since most cases are not asymptomatic.)

Solution: Model the people as nodes, and add edges between people if they are friends or close acquaintences. Color the sick nodes red and the healthy nodes blue. Search for a path connecting the patients that does not contain too many blue nodes.

Problem 7.b. Imagine you have a population in which some few people are infected with this weird virus. You also have a list of locations that each of the sick people were in during the last 14 days. Determine if any of the sick people ever met.

Solution: Model the people as nodes and the locations as nodes, and add edges between people and the locations they were in. Now look at any location node that has degree > 1.

Problem 7.c. You are given a set of jobs to schedule. Each job j starts at some time s_j an ends at some time t_j. Many of these jobs overlap. You want to efficiently find large collections of non-overlapping jobs so that you can assign each collection to a single server.

Solution: Each node is a job. Add an edge between to nodes if the respective jobs do not overlap. Now you need to find collections of nodes that are not neighbors. (This is referred to as an *independent set* and hence can be solved by algorithms for finding a maximal independent set. In fact, there are simple greedy solutions for the Interval Scheduling problem.)

Problem 7.d. An English professor complains that students in their class are cheating. The professor suspects that the cheating students are all copying their material from only a few different sources, but does not know where they are copying from. Students that are not cheating, on the other hand, all submit fairly different solutions. How should we catch the cheaters?

Solution: Each student's essay is a node. Add an edge between two nodes if the respective essays are similar. A cluster of nodes all connected to each other likely indicates cheating. (If they are all connected, this is about finding a clique. Otherwise, this is a clustering problem.)

Problem 7.e. There are n children and n presents, and each child has told you which presents they want. How do we assign presents to children?

Solution: Each child is a node. Each present is a node. Add an edge connecting a child to a present if that present is acceptable to that child. Now find a set of edges that do no share any endpoints. (This is called a matching.)

Problem 8.
(A problem lovely as a tree.)
(Relevant Kattis Problem: https://open.kattis.com/problems/flyingsafely)

Assume you are given a connected graph with n nodes and m edges as an adjacency list. (You are given n but not m; assume each adjacency list is given as a linked list, so you do not have access to its size.)

Give an algorithm to determine whether or not this graph is a *tree*. Recall that a tree is a connected graph with no cycles. (What if you want to find a cycle?)

Your algorithm should run in $O(n)$; particularly, it should be independent of m. Assume $O(n + m)$ is too slow.

Solution: Any connected, undirected graph containing n nodes and $> n-1$ edges has a cycle. Thus, simply count edges in the graph, stopping when you find at least n. Notice, though, that since the graph is given as an adjacency list, each edge is represented twice: in the list of both the source and the destination. Thus, if you find $> 2n-2$ edges in the adjacency lists, you know there is a cycle. Otherwise, there is not. If you want to find a cycle, run DFS. You will be guaranteed to see a cycle before you examine $n+1$ edges.

Problem 9.
(Gone viral.)

There are n students in the National University of Singapore. Among them, there are $n-1$ friendships. (In case you are unfamiliar with the concept of friendship, it means that two people are friends with each other.) Note that friendship is a symmetric relation, but it is not necessarily transitive.

Any two people in the National University of Singapore are either directly or indirectly friends. Formally, between any two different people x and y, either x is friends with y or there exists a sequence $q_1, q_2, \ldots, q_k$ such that x is friends with q_1, q_i is friends with q_{i+1} for all $i < k$ and q_k is friends with y.

It was discovered today that **two** people were found to have coronavirus in the National University of Singapore.

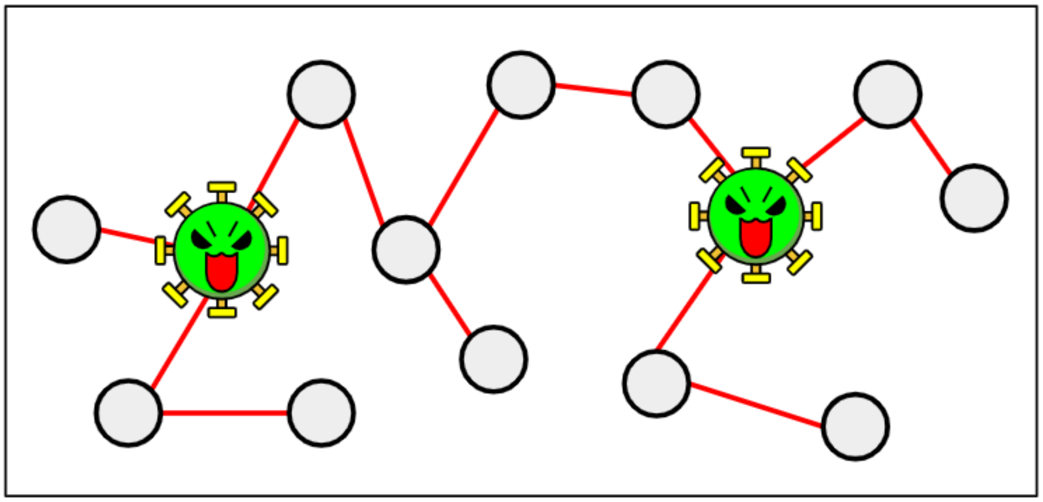


Figure 1: *Gone Viral.* (Matthew Ng Zhen Rui)

Every day, every person can meet with **at most one friend.** When these two people meet, if exactly one of them has the coronavirus, it will be transmitted to the other.

Give an $O(n \log^2 n)$ algorithm to determine the minimum possible number of days before it is possible that *everyone* has the coronavirus.

Solution: For the one infected student case, the solution is straightforward—consider the tree rooted at the infected student and define $f(x)$ as the minimum number of days needed for all students in the subtree of x to be infected assuming the x^{th} student is infected.
For leaf nodes $f(x) = 0$. For internal nodes, we have $f(x) = \max(f(c_1)+1, f(c_2)+2, \ldots, f(c_k)+k)$ where $c_1, c_2, \ldots, c_k$ are the children infected in this order. Since we want to minimize f, we should sort these $f(c_i)$'s so that $+1$ gets assigned to the maximum $f(c_1)$, $+2$ to the second-maximum and so on.
This can be done in $O(k \log k)$ for an internal node with k children. Overall it is $O(n \log n)$.
For the two infected students case, suppose the two infected students are x and y and consider the unique path from x to y, say $x, q_1, q_2, \ldots, q_t, y$. In the optimal solution, there exists some $0 \leq k \leq t$ such that such that $x, q_1, \ldots, q_k$ are infected by students originally infected by x and $q_{k+1}, \ldots q_t, y$ are infected by students originally infected by y. In other words, we can cut some edge along this path and compute the answers for the two trees (using the solution for the one infected person case). Suppose the tree containing x needs a days and the tree containing y needs b days. Then clearly if $a < b$ we should assign more students to x and otherwise y.
By binary searching on this path, we have an $O(n \log^2 n)$ solution.