Introduction to Algorithms	Problem Set 6 (3 problems)
CS 4820 Fall 2019	Due 11:59pm Thursday, October 31

There are three questions on this homework, a coding problem and two written questions. Your response to each written question needs to be at most two pages long.

Your homework submissions for written questions need to be typeset (hand-drawn figures are OK). See the course web page for suggestions on typing formulas. The solution to each written question needs to be uploaded to CMS as a separate pdf file. To help provide anonymity in your grading, do not write your name on the homework (CMS will know it's your submission).

Solutions to the coding problem need to be submitted to the **online autograder** at `https://cs4820.cs.cornell.edu/`.

Collaboration is encouraged while solving the problems, but

1. list the names of those with whom you collaborated with (as a separate file submitted on CMS);
2. you must write up the solutions in your own words;
3. you must write your own code.

Remember that when a problem asks you to design an algorithm, you must also prove the algorithm's correctness and analyze its running time. The running time must be bounded by a polynomial function of the input size.

(1) Scheduling Interviews I (10 points)

We are scheduling initial phone interviews for n job candidates that have applied for jobs at a company. The company has $k \leq n$ recruiters, and each recruiter is qualified to interview only some of the candidates. Candidates will be labeled $0, 1, \ldots, n-1$, and recruiters will be labeled $n, n+1, \ldots, n+k-1$. Each candidate needs to be assigned to a recruiter to be interviewed. We can represent the problem by a bipartite graph where nodes are the job candidates $0, 1, \ldots, n-1$ on one side, and recruiters $n, n+1, \ldots, n+k-1$ on the other side, and a job candidate i is connected to a recruiter j if j can interview i. We also want to make sure that recruiters are not overloaded, which means that for each recruiter j, we have c_j, a maximum number of interviews she could have. A valid schedule is a way to assign candidates to recruiters so that each candidate is assigned to a recruiter, and no recruiter j has more than c_j interviews scheduled (which is called recruiter_capacities$[j]$ in the code below).

Someone has already tried to find an assignment of candidates to interviewers, but they are having trouble. They have a valid assignment where each candidate i is assigned to recruiter preliminary_assignment$[i]$ (without overloading any recruiter). Unfortunately, preliminary_assignment$[n-1]$ is blank, and they are having trouble filling this last entry. In this problem, you are asked to use preliminary_assignment to decide if the there is a perfect schedule. If no such schedule is possible, we ask you to output the list of recruiters j such that if j is willing to interview one extra candidate the assignment will exists.

Specifically, you must code an efficient algorithm in Java that has the following behavior:

- If there exists a valid assignment that assigns all job candidates to recruiters, output it. Note that **there may be more than one such valid assignment**. We will accept any, as long as it assigns all job candidates to recruiters in a valid way (i.e., no recruiter is overbooked and recruiters only interview candidates they are qualified to interview).
- If no such assignment exists, you plan to ask one of the recruiters j to increase their capacity recruiter_capacities$[j]$. Output the list of recruiters j such that if their capacity is increased by 1 (while the other capacities remain the same), then a solution will exist.

The time limit of this problem will be 2 second. Your algorithm must run in $O(m)$ time, where m is the number of edges in the graph. Solutions that take longer (e.g., $O(nk)$) will only get partial credit.

We illustrate the problem with the following **example**:

- Suppose we have $n = 3$ candidates, $k = 2$ recruiters, candidate neighbors neighbors$[0] = (3, 4)$, neighbors$[1] = (3)$, and neighbors$[2] = (3)$ (the rest of the array can be inferred from these entries), and recruiter capacities recruiter_capacities$[3] = 2$, recruiter_capacities$[4] = 1$. If we are given preliminary_assignment $= (3, 3, \cdot)$, then a fully satisfying assignment does exist: it is valid_assignment $= (4, 3, 3)$.
- For the (otherwise) same input, if recruiter_capacities$[3] = 1$, recruiter_capacities$[4] = 2$, and preliminary_assignment $= (4, 3, \cdot)$, then no fully satisfying assignment exists, and the only way to create one is to increase recruiter_capacities$[3]$.

Starter code provided. Unlike in previous problems, we have set up a partial code for you that reads the input and writes the output, and sets up useful data-structures, and all you have to do is add the missing part: deciding if an assignment exists or which recruiters can help. Our partial code is available on CMS, called *public_code.java*. You need to add your code between the lines "YOUR CODE STARTS HERE" and "YOUR CODE ENDS HERE".

Note that **you will not parse input nor print output**. We parse the input and provide you with complete and incomplete data structures. You will use the completed data structures to fill the incomplete data structures. Our framework with use the data structures you fill to format and print the output. We describe these data structures below, and also describe the input/output file formats.

Complete data structures we provide:

- The arraylist neighbors, where neighbors$[i]$ is an arraylist all people who can be matched with i. That is, if if $i = 0, 1, \ldots, n-1$ then neighbors$[i]$ is an arraylist of the indices of the recruiters qualified to interview candidate i. If $i = n, n+1, \ldots, n+k-1$ then neighbors$[i]$ is an arraylist of the indices of the candidates that recruiter i is qualified to interview.
- The array recruiter_capacities, where recruiter_capacities$[j]$ is the limit of candidates that recruiter j can interview. Note that because of our indexing scheme, entries 0 through $n-1$ of this array are empty
- The array preliminary_assignment, where preliminary_assignment$[i]$ is the recruiter assigned to interview the i^{th} job candidate in the provided first attempt, for $i = 0, 1, \ldots, n-2$. Recall that preliminary_assignment$[n-1]$ is blank, and they are having trouble filling this last entry.

Data structures we provide that you must fill in:

- The boolean value exists_valid_assignment, which you should set to true if there exists a way to assign every job candidate to a recruiter without increasing their limits (and false otherwise).
- The array valid_assignment, which you should populate *only if* you set the previous variable to true. valid_assignment$[i]$ is the index of the recruiter assigned to candidate i in your full valid assignment. (Note that i ranges over $0, 1, \ldots, n-1$, and the values of the array range over $n, n+1, \ldots, n+k-1$.)
- The array bottleneck_recruiters, which you should populate *only if* you set exists_valid_assignment to false. bottleneck_recruiters$[j]$ should be set to 1 if increasing the limit of recruiter j by 1 (while leaving the other limits the same) would make a full assignment possible, and 0 otherwise. (Note that the array is defined over $j = n, n+1, \ldots, n+k-1$, so **the first n entries must be set to 0.**)

Input file format (just FYI; only work with the data structures above):

- The first line has two numbers, n and k, the number of candidates and recruiters.
- In the i^{th} line of the next n lines (for $i = 0, \ldots, n-1$), there is a number indicating how many recruiters candidate i can interview with, followed by the indices of those recruiters. The indices appearing in the i^{th} line are stored in neighbors$[i]$. (neighbors$[j]$ for $j \geq n$ are inferred from these lines, as well.)

- The next line has k numbers representing the capacities of the recruiters. We store the i^{th} number in recruiter_capacities$[n+i-1]$.
- The last line has $n-1$ numbers representing the preliminary assignment. We store the i^{th} number in preliminary_assignment$[i-1]$.

Output file format (just FYI; only work with the data structures above):

- If there exists a full assignment, the first line will be "Yes", and the second line will be the n entries of valid_assignment.
- If there does not exist an assignment, the first line will have the string "No", and the second line will be the $n+k$ entries of bottleneck_recruiters (where the first n entries are 0).

The only libraries you are allowed to `import` are the ones in `java.util.*`

Warning: be aware that the running time of calling a method of a built-in Java class is usually not constant time, and take this into account when you think about the overall running time of your code. For instance, if you used a LinkedList, and use the `indexOf` method, this will take time linear in the number of elements in the list.

We have set up an **online autograder** at `https://cs4820.cs.cornell.edu/`. You can upload solutions as many times as you like before the homework deadline.[1] **You need to have the main method of your code named Main, must not be "public", must not be part of a package**. When you upload a submission, the autograder will automatically compile and run it on a number of public test-cases[2] that we have prepared for you, checking the result for correctness and outputting any problems that may occur. You should use this facility to verify that you have interpreted the assignment specification correctly. After the deadline elapses, your last submission will be tested against a new set of *private* test cases, which your grade for the assignment will be based on.

(2) Scheduling Interviews II. (10 points) In Problem 1 above, we were assigning candidates to interviewers for a company without scheduling times for the interviews. In this problem, you are asked to set up a more detailed schedule. As in the previous problem, you are scheduling initial phone interviews for n job candidates at a company, with $k \leq n$ potential recruiters. Each candidate needs only one interview for now: concretely, each candidate needs to be matched to exactly one recruiter to be interviewed. The candidates are applying for different kinds of jobs, and recruiters are qualified to interview candidates only for some of the jobs. For each candidate i, you are given a list L_i of the recruiters who are qualified to interview that candidate. You also don't want to overload the recruiters, so each recruiter j has a limit c_j of the maximum number of interviews he or she can do.

Each interview will take one hour, and will be scheduled in one of T possible non-overlapping one-hour time slots $h_1, \ldots h_T$, each starting at the top of the hour (e.g. 10 AM, 1 PM, 5 PM ...). In addition to the input above, each interviewer and each candidate is asked to list the hours that they would be able to interview or be interviewed. Please note that each interviewer is asked to list strictly more than c_j possible slots to make scheduling easier.

Give an algorithm that finds an interview schedule if one exists. Let $m = \sum_i |L_i|$. Your algorithm should run in time polynomial in n, k, m, and T.

(3) Reduction. (10 points) An *Independent set* in an undirected graph $G = (V, E)$ a set of vertices $I \subset V$ with no two vertices $v, w \in I$ connected by an edge, that is $(v, w) \notin E$. The Independent Set problem has input an undirected graph G and an integer k and asks to decide if G has an independent set of size k. This a famous hard problem will be discussed in the lecture on Monday Oct 28 (see also section 8.1 of the book).

[1] The deadline shown on the auto-grader is the latest time at which you will be allowed to submit your code, including the maximum number of slip days. If you submit your code past the actual deadline, you will automatically be charged slip days accordingly, so please be careful.

[2] They will also be available on CMS.

Your friend likes getting together small groups of extremely independent people. He claims to have written a super-fast solver for the following graph problem to help him do this: He mapped all people he ever considered inviting as an undirected graph $G = (V, E)$ with the set of nodes representing people, and two nodes $v, w \in V$ connected by an edge if they regularly talk. He views two people $v, w \in V$ as *very independent* if not only they are not connected by an edge, but also they have no shared neighbor u, that is, no node u with edges (uv) and (uw) in E (the reason is that such a u may be influencing both v and w, which does make the two of them less independent). When he is arranging a meeting, he identifies a subset $S \subset V$ of potential candidates, and then looks for k people in S that are very independent. So the VeryIndependent Set problem is given an undirected graph $G = (V, E)$, a subset of nodes $S \subset V$, and an integer k decide if there is a very independent set $I \subset S$ of size k.

You are impressed by what your friend's code can do, and wonder if this code is useful to solve the famous Independent Set Problem also. Show that given an undirected graph G and integer k for the Independent Set problem, you can solve this problem by a single call your friend's code. It is okay to modify G to a new graph G' before calling your friend's code as long as G' is polynomial in the size of G. In providing a solution to this problem you need to do the following:

- provide a polynomial time algorithm that converts an input to the Independent Set problem, an undirected graph $G = (V, E)$ and an integer k to an input to the VeryIndependent Set problem, which consists of a graph $G' = (V', E')$, $S' \subseteq V'$ and integer k'
- prove that $G = (V, E)$ has an independent set of size k then G' has a very independent subset of the set S' of size k'
- and prove that if $G = (V, E)$ has no independent set of size k then G' also doesn't have a very independent subset of the set S' of size k'

Note that the last two proves show that the independent set problem on G is equivalent to the very independent set problem on G'.

(Your solution will show that Independent Set $\leq$ VeryIndependentSet, and in the coming week, we'll show the Independent Set problem is NP-complete.)