3.1 Assigning Groups

In a group of people, it is to be expected that some of them may not want to work with each other. Assuming that each person has at most d other people that they don't want to work with, divide the people into $d+1$ groups so that everyone is in exactly one group and no one is in a group with someone they don't want to work with.

> Sections (like this one) which marked with a vertical line on the left side are commentary — discussions about the algorithm development process that wouldn't be part of a writeup.

Establish the problem.

- *Specifications.*

  Task: assign people to $d + 1$ groups so that each person is in exactly one group and no one is in a group with someone they don't want to work with

  Input: n people, and for each, the up to d other people they don't want to work with

  Output: an assignment of people to groups (either a which-group label for each person or the membership for each group)

- *Examples.*

  > Examples illustrate and clarify the specifications, including handling in special cases. When first establishing the problem, look for ambiguities, points of confusion, or cases that might require special handling. These things may also be uncovered later — come back and add examples later as the need is discovered.

  Is not wanting to work with someone mutual? That is, if person A doesn't want to work with person B, does person B always also not want to work with person A? — Assume yes, since otherwise $d + 1$ groups might not be sufficient even if there are at most d people any given person doesn't want to work with. (For example, consider three people (A, B, C) where A doesn't want to work with B, B doesn't want to work with C, and C doesn't want to work with A. $d = 1$ in this case, but everyone needs to be in their own group.)

  Does "divide into $d + 1$ groups" mean exactly $d + 1$ groups or at most $d + 1$ groups? Do there have to be $d + 1$ groups? (Can some groups be empty?) — Groups can be split as needed at the end, since removing people from a group without conflicts isn't going to introduce conflicts. However, no criteria are given for how this might be done (create a bunch of one-person groups, or balance group size, or ...?) so we'll assume that the goal is at most $d + 1$ groups.

Identify avenues of attack.

> Keep in mind that the goal of these steps is *not* to develop the algorithm, but to start to explore (or rule out) possible directions.

- *Known targets.*

  > Not really applicable here — we aren't given any time or space requirements, and since every person needs to be assigned to a group, we can't do better than $O(n)$.

- *Paradigms and patterns.*

  > Consider how the iterative patterns would apply to this problem.

  Process input: for each person, assign them to a legal group.

  > With the output viewed as the membership for each of the groups, the output elements are the groups so the produce output approach is to determine each group in turn. If the output is viewed as a labeling of the input elements — each person is assigned a group — then the next output

> element is a label for the next input element and the produce output approach reduces to the same thing as the process input approach. So we'll go with the first.

Produce output: for each group, determine who is in the group.

Narrow the search space: not applicable, this isn't a search problem.

Define the algorithm.

- *Main steps.*

> Both process input and produce output patterns are applicable. In this case, it seems like it might be easier to decide which group to put someone in than who all belongs in a group, so let's go with process input.

```
for each person
  add them to a group that doesn't have someone they don't want to work with
```

> This follows the process input pattern, filling in enough details to be an algorithm. ("Assign to a legal group" lacks specifics about what constitutes "legal".)

- *Exit condition.*

> For the produce input pattern, the exit condition has the form "when all of the input elements have been processed".

When all of the people have been assigned to groups.

- *Setup.*

Create d+1 empty groups.

- *Wrapup.*

The assignment of people to groups is the desired result. Depending on the implementation and the desired format of the output (labels with the group number associated with people, or the membership of each group), it may be necessary to reformat the solution.

- *Special cases.*

None.

- *Algorithm.*

`algorithm groups(people,prefs)` —

> Task: assign people to $d + 1$ groups so that each person is in exactly one group and no one is in a group with someone they don't want to work with
>
> Input: n people, and for each, the up to d other people they don't want to work with
>
> Output: an assignment of people to groups (either a which-group label for each person or the membership for each group)
>
> ```
> create d+1 empty groups
> for each person
>   add them to a group that doesn't have someone they don't want to work with
> ```

Show termination and correctness.

- *Termination.*
  - *Measure of progress.*

> For the process input pattern, the measure of progress is the number of input elements processed.

The number of people assigned to groups.

- *Making progress.*
  Every loop iteration assigns another person to a group, increasing the number of people assigned to groups.
- *The end is reached.*
  Repeatedly increasing the number of people assigned to a group means that eventually all will have been assigned to a group.

- *Correctness.*
  - *Loop invariant.*
    > For iterative algorithms, the loop invariant often takes the form of an assertion that the solution so far is correct. For the process input pattern, the solution so far is after k input items have been processed. Be sure to include all of the properties needed for a correct solution.

    After the first k people have been assigned to groups, no one is in a group with someone they don't want to work with and there are (at most) $d + 1$ groups.
  - *Establish the loop invariant.*
    > Explain why the loop invariant holds at the beginning of the first and second iterations of the loop (i.e. before and after the first iteration).

    d+1 groups are created at the beginning and the loop body does not create any new groups, so there will never be more than d+1 groups.
    Before the first iteration ($k = 0$). No one has been assigned to a group, so no one can be in a group with someone they don't want to work with.
    After the first iteration / before the second iteration ($k = 1$). The first person can be assigned to any group because all of the groups are empty, and thus no group has someone the first person doesn't want to work with.
  - *Maintain the loop invariant.*
    > Explain why the loop invariant continues to be true after each iteration — assuming that it holds after iteration k (just before iteration k+1), explain why it also holds after iteration k+1 (just before iteration k+2).

    Assume that the invariant holds after k i.e. the first k people have been assigned to groups so that no one is in a group with someone they don't want to work with, and there are at most $d + 1$ groups.
    The next iteration assigns person k+1 to a group. Since there are at most d people that person k+1 doesn't want to work with and d+1 groups, there will always be at least one group that person k+1 can be added to (*). Furthermore, the loop body does not create any new groups so there are still only d+1 groups.
    (*) This assumes that "don't want to work with" is mutual — if there are people that a person is OK working with but who don't want to work with them, there could be more than d people person k+1 has to avoid. There is no solution in that case.
  - *Final answer.*
    > Explain why the whole algorithm — setup, loop, wrapup — means that the final result is a correct answer to the problem.

We've now shown that the loop invariant holds before and after every iteration — including after the last iteration, when the exit condition becomes true. The exit condition is that all (n) people have been assigned to groups, so the loop invariant gives us that after the first n people (i.e. all n people) have been assigned to groups, no one is in a group with someone they don't want to work with and there are d+1 groups...exactly what is required.

Determine efficiency.

- *Implementation.*

  > Identify data structures and, as necessary, specific implementations of those data structures to efficiently support the algorithm. Also fill in any algorithmic details that are needed in order to establish the running time.

  The format for the input isn't specified, but it is reasonable to assume that it is possible to iterate through the collection of people in $O(1)$ per person ($O(n)$ total). ($O(n)$ traversal applies to many types of collections.) For the "don't want to work with" preferences, a hashtable-based Map storing a list of each person's conflicts supports $O(d)$ "for each person the current person doesn't want to work with..."

- *Time and space.*

  > Assess the running time and space requirements of the algorithm given the implementation identified.

  A brute force way to find a legal group for a person is to consider each group in turn and, for each group, go through the person's list of conflicts to see if any of those people is in the group. This is $O(d^2)$ per person (d groups and $O(d)$ go-through-the-conflicts) assuming a $O(1)$ group membership check (which is possible with a hashtable-based Set).

  Adding a person to a group is $O(1)$ with a hashtable-based Set.

  Putting this all together results in $O(d^2n)$ time.

- *Room for improvement.*

  > The avenues for improvement for an iterative algorithm are to reduce the number of iterations of the loop and/or to reduce the work done in each iteration.

  Since we have to assign each person to a group, we're not likely to reduce the number of iterations of the outer loop.

  It's not possible to improve on an $O(1)$ add-to-group, so any improvements have to come from a more efficient way to find a legal group for someone.

  > To continue this line of thought, consider targets to aim for — $O(1)$ to find each person's group is probably too optimistic, but we might hope for $O(d)$. This would be possible with an $O(1)$ determine-if-person-p-can-join-group-g operation. How do we achieve $O(1)$? Through lookup (or storage) rather than computation. This means we'd need to store something like a boolean array for each person — slot g of the array would be true if the person could join group g and false otherwise.
  >
  > How would this information be initialized and updated? That's something to work out, but we have something concrete to think about that might lead to an improved implementation.