3.1 Assigning Groups

> 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.

> Defining — and understanding — the precise task to be solved is an essential prerequisite for algorithm development.

- *Specifications.*

  > State complete specifications for the problem. What is the problem? What do you start with (input) and what is the end result (output)? What are the legal input instances and the required output for each?

  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.

  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.*

  > If needed, give examples (specific inputs and the corresponding outputs) of typical and special cases to clarify the specifications.

  > Examples are meant to clarify the specifications, including handling in special cases. In thinking about clarification examples, ambiguities in the specifications may be identified.

  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? — 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.

> Determine the algorithmic approach(es) to try.

- *Targets.*

  > Identify any applicable time and space requirements for your solution.

> We aren't given any targets here, so...

With n people to assign to groups, it seems like any algorithm will be $\Omega(n)$ and thus $\Theta(n)$ would be a good goal.

- *Patterns.*

  > Consider the iterative patterns.

  > There are three main patterns: process input, produce output, and narrow the search space. The goal in this step is to think about how this task might fit into the patterns — we're not trying to come up with an algorithm, just to identify what pieces need to be sorted out to complete an algorithm in a particular vein.
  >
  > Process input is potentially applicable when the input is a collection of elements — for each input element, process it. Produce output is potentially applicable when the output is a collection of elements or can be built up incrementally — repeatedly add the next thing to the output. Narrow the search space is potentially applicable when there is a convenient representation of the search space and a way to eliminate non-solutions without examining them directly.

  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: repeatedly determine the membership for the next group.

  > Narrow the search space: the goal here is an assignment to groups, so to view this as a search problem means looking for a legal assignment to groups in the space of all possible ways to assign n people to up to $d + 1$ groups. This is a very large and rather unwieldy search space to think about how to capture, so narrowing the search space is not a good pattern to try to pursue.

Define the algorithm.

> Assemble the pieces of the algorithm according to the template for the particular paradigm and pattern, culminating in a statement of the algorithm itself.

- *Main steps.*

  > This is the core of the algorithm — the loop body. What's being repeated?

  > Both process input and produce output approaches are plausible here, so pick whichever seems easiest to figure out as a starting point. With produce output, there's the additional question of when to stop assigning people to one group and move on to the next, while process input seems very straightforward — just (somehow) decide on a group for each person. If it is a tossup, go with process input. But the important thing here to pick something to try — you don't generally know in advance which approach will work (and both may be possible and lead to different algorithms), so just back up and try something else if you hit a dead end rather than worrying about avoiding the dead end in the first place.
  >
  > So let's go with process input. How to assign a person to a group? Focus on the criteria for picking a group — keep it as simple as possible and don't dismiss an idea just because it might seem inefficient to implement. Start with a correct algorithm, then worry about efficiency — it may be possible to find a more efficient implementation than it initially appears, or maybe a different algorithmic approach *is* needed for an efficient solution, but in that case having an algorithm gives you a starting point.

```
for each person
assign the person to the first group that doesn't have someone they don't
want to work with, creating a new group if there is no such group already
```

> Why the first group specifically? We need a legal group, but there might be more than one of those so we need to know how to break ties. Choosing the first legal group means we can stop as soon as a legal group has been found. By contrast, choosing a random group would also result in a legal assignment but would require finding all of the legal groups instead of just one.

- *Exit condition.*

  > When does the loop end?

  > 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.

  > Legality of the solution is not part of this — it's not "when all of the people have been assigned to legal groups". We're done when everyone has a group; that the assignments adhere to the rules is the responsibility of the loop body and how the groups were chosen.

- *Setup.*

  > Whatever must happen before the loop begins.

  Initially there are no groups.

- *Wrapup.*

  > Whatever must happen to get the final answer after the loop ends.

  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.*

  > Make sure the algorithm works for all legal inputs — identify the cases that need to be handled and address how that handling is incorporated into the previous steps (if not already accounted for).

  > Typical special cases to consider for collections are empty collections and collections with a single element. Also consider if particular values could cause problems and make sure any clarification examples are handled.

  $n = 0$ and $n = 1$ — This just affects the number of loop iterations. No special handling needed.

  B doesn't want to work with A, but A is fine with B — Not possible given our assumptions, no special handling needed.

  Do we have to have $d + 1$ groups? — No, given our assumptions so no special handling needed.

- *Algorithm.*

> Assemble the algorithm from the previous steps and state it.
>
> There shouldn't be new elements here, instead bring together the main steps, exit condition, setup, and wrapup along with any handling needed for special cases and state the whole 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)

```
for each person
  assign the person to the first group that doesn't have someone they don't
   want to work with, creating a new group if there is no such group already
```

Show termination and correctness.

> Show that the algorithm produces a correct solution.

- *Termination.*
  > Show that the loop — and thus the algorithm — always terminates.
  - *Measure of progress.*
    > Identify a quantity and the direction of change that leads towards the exit condition.
    >
    > For the process input pattern, this is the number of input elements processed.

    The number of people assigned to groups. (increasing)
  - *Making progress.*
    > Explain why every iteration of the loop advances the measure of progress towards the exit condition.

    Every loop iteration assigns another person to a group, either an existing group or a new one.
  - *The end is reached.*
    > Explain why making progress ensures that the exit condition is always reached.

    Increasing the number of people assigned to a group means that eventually all will have been assigned to a group.
- *Correctness.*
  > Show that the algorithm is correct.
  - *Loop invariant.*
    > State a loop invariant.
    >
    > For the process input pattern, this typically takes the form of "we have a correct solution for the first k input elements".

    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.
  - *Establish the loop invariant.*

> Explain why the loop invariant holds at the beginning of the first and second iterations of the loop.

At the beginning of 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, and there are at most $d+1$ groups ($0 < d + 1$ for $d \geq 0$).

At the beginning of the second iteration ($k = 1$). Only one person has been assigned to a group, so no one can be in a group with someone they don't want to work with, and there are at most $d + 1$ groups ($1 \leq d + 1$ for $d \geq 0$).

- *Maintain the loop invariant.*

  > Explain why the loop invariant continues to be true after each iteration — assuming that it holds at the beginning of iteration k, explain why it also holds at the beginning of the next iteration $(k + 1)$.

  Assume that the invariant holds for k: 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.

  Show that the loop invariant still holds after the next iteration, that is, after the next person has been added to a group, the first $k + 1$ 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.

  No one is in a group with someone they don't want to work with. The algorithm assigns person $k + 1$ to the first group which doesn't have someone they don't want to work with. There is such a group because a new group is created if needed. Given the assumption that "doesn't want to work with" is mutual, there can't be someone in the group person $k + 1$ is assigned that doesn't want to work with person $k + 1$ because person $k + 1$ would also not want to work with them.

  At most $d+1$ groups. Either person $k+1$ was assigned to an existing group, or they were assigned to a new group. Assigning person $k+1$ to an existing group doesn't change the number of groups, so if there weren't more than $d + 1$ before, there won't be now. Assigning person $k + 1$ to a new group does increase the number of groups, however there can't be more than d existing groups that person $k+1$ can't join because there are at most d people person $k+1$ doesn't want to work with. As a result, there can't be more than $d + 1$ groups including the new group.

- *Final answer.*

  > Explain why the whole algorithm — setup, loop, wrapup — means that the final result is a correct answer to the problem.

The loop invariant states that 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 loop exits when all of the people have been assigned to groups, that is, when $k = n$. Since the loop invariant is true every time the exit condition is tested, we have that when the loop exits, all n 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 — as required.

Determine efficiency.

> Evaluate the running time and space requirements of the algorithm.

- *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 an array or List of people, an array or List for each person's "don't work with" preferences, and a $O(1)$-time way to get the "don't work with" list for a particular person. Traversal of an array or List is $O(1)$ per element, and adding an element to an unsorted array or List is $O(1)$.

Set membership and insertion are $O(1)$ expected time for a hashtable, but traversal of a hashtable is $O(N)$.

We need to specify how to carry out the step

```
find the first group that doesn't have someone they don't want to work with
```

"Find the first group" suggests going through the groups one-by-one and determining whether person $k+1$ can be added to that group:

```
for each group g,
  determine if there's someone in group g that person k+1 doesn't want
   to work with
```

There are a couple of strategies for determining whether a person $k+1$ can be added to a particular group g:

- For each person on person $k+1$'s "don't work with" list, are they in group g?
- For each person in group g, are they on person $k+1$'s "don't work with" list?

- *Time and space.*

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

  Consider the two strategies identified above for determining whether person $k+1$ can be added to group g:

  - *For each person on person $k+1$'s "don't work with" list, are they in group g?* There are up to d people on the "don't work with" list, so this means $O(d)$ membership checks. Repeated for all (up to $d+1$) groups means $O(d^2)$ membership checks to find a group for person $k+1$.
  - *For each person in group g, are they on person $k+1$'s "don't work with" list?* We don't know how many people are in each group, but can observe that there are a total of k people in all of the groups so far, so this means $O(k)$ membership checks to find a group for person $k+1$.

  To assign all n people to groups means a total of $O(d^2n)$ or $O(n^2)$ membership checks plus up to $d+1$ create-new-groups and n add-person-to-groups. With an array or List for the people to assign to groups and each of the "don't work with" lists and a Set (hashtable) for each group, the first strategy means a total running time of $O(d^2n) + O(d+1) + O(n) = O(d^2n)$. With a hashtable Set for each "don't work with" list, the second strategy means a total running time of $O(n^2) + O(d+1) + O(n) = O(n^2)$. (If the "don't work with" lists aren't provided as hashtable Sets, the sets can be built by traversing each of the "don't work with" lists. This is $O(dn)$, which doesn't change the overall $O(n^2)$ for the whole algorithm.)

- *Room for improvement.*

  > Are the targets met? Is it necessary to do better? If improvements in running time and/or space are needed, identify possible avenues 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.

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

  Instead focus on how long it takes to find a group for the current person — $O(1)$ 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? Before anyone is assigned to a group, every group is legal for every person — initialize all of the arrays to *true*, which is $O(dn)$ because there are at most $d+1$ groups and there are n people. When person p is added to group g, slot g would have to be updated for everyone who doesn't want to work with person p — up to d people, making an $O(1)$ add-person-to-group operation into an $O(d)$ operation. However, this means that for each person, it is $O(d)$ to find a group and $O(d)$ to add them to the group and update everyone who doesn't want to work with them — a total of $O(dn)$. Combined with the time to initialize the arrays, teh total running time is $O(dn)$ — an improvement over the previous strategies.