COSC 1P03 — Assignment 1 — Weee are the Champions...

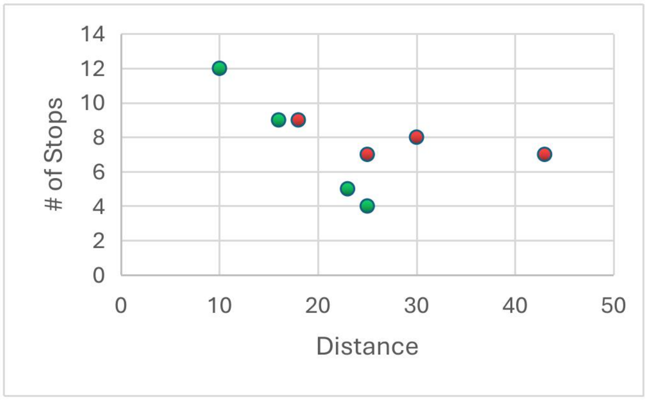
Multiobjective optimization can be a tricky thing sometimes. Firstly, one has to know what 'multiobjective optimization' is! Let's say you wanted to deliver some packages to multiple homes. You might want to prioritize using the least amount of gas, or you might want to complete your deliveries in the least amount of time. The two aren't always the same thing, right? Each is an objective. Multiobjective optimization is just about trying to find solutions to problems that consider multiple metrics for 'good'.
We're not going to bother with the optimization part (right now). We're just going to look at how we can qualify 'good', for multiple possible dimensions (different objectives).

Vilfredo Vanguard

Sometimes optimization means minimization; other times maximization. In other words sometimes you want to keep numbers as low as possible, or as high. For simplicity, we'll stick with the most common: 'lower is better'. e.g. maybe you want to plan a trip flying, and you'd like it to be as fast as possible, but separately want the fewest layovers.
If you had two candidate routes (say, one that only took 7 hours but had 3 layovers; and another that was 12 hours but no layovers), there might simply be *no* definitive 'best answer'. We might not force it.
But at the same time, other solutions can be objectively worse (13 hours with 4 layovers, or even just 8 hours with 3).
One possibility that's sometimes used is a *weighted sum*: pick some multiplier for each dimension, multiply by each individual value, and add it up. Compare those weighted sums for each candidate to name the 'best'. But among other problems an immediate issue is one of bias: it tries to compress multiple objectives into one, and inevitably favours one over the other. What's the alternative?
A **Vilfredo Vanguard** (sometimes a Vilfredo *Edge*, but that doesn't have the same alliterative charm) is a subset of data readings that have *unconquered attributes*. They can be pretty confusing at first, but I find visualizations make this make a *lot* more sense:

Distance	Stops
23	5
43	7
18	9
30	8
25	4
16	9
10	12
25	7

We have multiple possible routes to consider (what we'll refer to as 'candidate solutions').
Above we have some possible tuples of results for eight different candidate solutions. And *clearly* it's easy to tell which ones should be considered and which ignored, right? No? How about now:



I've highlighted four readings as red, and 4 as green. (Left is 'good'; Down is 'differently good')

- The green ones each have *at least one axis* in which nothing else beats them
    - e.g. the upper-left one has a terribly high 12 stops, but it's also the shortest distance
    - The bottom-most has a middling distance, but the fewest stops
    - The second pip isn't the shortest distance, but *is* the shortest distance for anything with fewer than 12 stops
- So what about the red ones? The red ones have objectively better alternatives:
    - The leftmost red one has a distance of 18 and 9 stops. There'd be no benefit in choosing that over the one that also has 9 stops, but achieves it in a distance of 16
    - The second red pip has [25:7], but the [25:4] below it would already be better, and the [23:5] is better than it in *both* dimensions

The VV Algorithm

We want to isolate those potential results comprising the Vilfredo Vanguard; there are two aspects to this:

1. Determining whether or not an individual point is unconquered
2. Keeping track of which are/aren't, without making it complicated

Generally we could store each candidate solution as e.g. its own class, but since we've just learned about arrays, let's use something that looks like our table of values above!

- We'll have a separate row for each total candidate solution
- Each row will have a number of columns matching our number of axes
    - In general, this value *could* get large, but remember: it's still a 2D array, regardless!

Here's the basic algorithm itself:

- For each candidate solution (each row in the array)
    - Check if it is conquered by *any* other candidate solution
        - Candidate A conquers Candidate B when A contains at least one value less than the corresponding value in B *and* has no other values where B's is lower
        - So if all values are identical, neither conquers the other
        - If A contains values lower than B's and B contains values lower than A's, neither conquers the other
        - Put another way: if all values are identical or A contains even a single lower value, at the very least A cannot be conquered by B
    - If it is conquered, it's not part of the Vilfredo Vanguard; if unconquered, it's part of the VV

Generally, we're not looking for conquering rows; we're looking for rows that aren't conquered.

Assignment Task

You've been provided with `simpleData.txt`, to reflect the example shown above. Its first line is the number of columns (2), the second is the number of rows/entries (8), and then one row for each entry (tab-separated for the two columns). It should be easy loading this via `ASCIIDataFile`.

Important: this is just *one* data file! Your eventual program will support *any*, with any number of objectives! So long as it's formatted correctly, I should be able to load it! (***Without*** recompiling!)

What you need to do is to open *any* such data file, and report to the user on the following:

- What candidate solutions are being considered
- What candidates are part of the Vilfredo Vanguard

So for the provided sample data, you might get something like:

```
For data entries:
 [23|5] [43|7] [18|9] [30|8] [25|4] [16|9] [10|12] [25|7]
Vilefredo Vanguard:
[23|5]
[25|4]
[16|9]
[10|12]
```

Again, the point isn't the outcome. *We already know the answer this time*.

The point is developing the techniques for solving this suitably.

Tips

Remember: the sole reason for assignments is their pedagogical value in strengthing core skills. Putting it off and/or trying to slap together 'a solution' defeats the purpose. Treat this as a *collection* of individual tasks, each of which you need experience with!

Here's the suggested 'plan of action':

1. Re-read this entire assignment all over again. Yes, really
2. Take notes of all the requirements
   a) Where is data coming from? Where is it going? What does the data really *mean*?
   b) What should the final program do?
   c) What are the pieces that build up to make that final program?
3. Re-read this entire assignment all over again, again. Yes, still really
4. Ideally, work out how you're going to code *all* of this *before* touching your computer
5. Start working on the code:
   a) Start up IntelliJ and create a project. Create a main class (say, `VilfredoVanguard`), give it a main method, and create its Run Configuration within IntelliJ
   b) Write a function to open an ASCIIDataFile, find out the required dimensions for that experiment, allocate an appropriately-sized 2D array, load the data into said array, and then return that array
   c) Write a method for displaying the contents of a provided 2D array, just to ensure that what you get matches the table above. If it doesn't, **stop here and get it working**! Trying to proceed without this is a declaration that your time has no value, and you need to start being nicer to yourself than that!

d) Write a function to accept either two 1D arrays, or two row indices and a 2D matrix, and identify whether the first specified row is conquered by the second row
   - The reason you can go with either approach is a 2D array's just an 'array of arrays'. Each row is a candidate, so if you go with indices and the matrix you just get two rows anyhoo
   - Work on 'proof by contradiction'. First assume it's *not* conquered, and then consider each dimension.
     - ❖ If you find a single dimension where the target candidate is *lower* than the other, you immediately know it's not conquered
     - ❖ If they're all equal, it's not conquered by the other
     - ❖ If the other candidate has at least one value lower than the corresponding value for the target candidate, then the target will be conquered unless you eventually find a lower value within the target than the other's equivalent value
   - **Definitely** feel free to ask about this before our lecture!

e) Write a function to accept candidate solution (row or index) and a matrix, and identify whether that candidate is conquered by any other member of the matrix
   - Again, proof by contradiction. Assume there's nothing that conquers it. Iterate over the matrix; if you find a counter-point, then it is. If you get through them all without finding something that conquers it, it's unconquered

f) Write a function to identify the Vilfredo Vanguard of the data. To be clear, this should create, populate, or otherwise use an array for this step. How? Up to you!

g) Write a method to report to the user on the Vilfredo Vanguard. By this point we care about the candidate solutions; not row indices or other organizational stuff

h) Ensure the user may select *any* (suitably-formatted) data file
   - To be clear: your marker will not bother cleaning up your mess if you try hard-coding this
   - Remember: use the ASCIIDataFile for reading; not File/Scanner/etc.

If all of that seems like 'a bit much', it *kinda* is.
But technically it's probably still the fastest way to do it.

Submission Requirements

First we'll cover the submission instructions, and then we'll cover 'the other stuff'.
To submit, bundle up your IntelliJ project (which, remember, you developed on a single computer) into a .zip file. Make sure to include the whole project — nothing more, nothing less.
Include a sample execution, but that should just be a formality so long as you're following instructions.
Even though a sample data file is provided, it's advisable to still include it to make it quicker for the marker to evaluate (before moving on to *other* data files).

Reminders

- Don't submit videos, or entire copies of the JDK
- Don't submit fragments of multiple projects
- Don't submit just .java or .class files; whole projects please
- Remember: JDK 11, all the way through!