CITS2200 Project 2022

This project is marked out of 40, and is worth 20% of your final unit mark.

You may work on this project individually or in pairs.

A Note on Academic Conduct

While you are encouraged to do research to learn about data structures and algorithms, please remember that by university policy, anything you submit for assessment must be your own wholly original work.

If you do refer to external sources, make sure to reference them in your work (a link is fine). Make sure that you do not (intentionally or unintentionally) copy any part of the source you are referring to into your submission. Every year we have a shocking rate of academic misconduct among project submissions, so be mindful to keep the work your own.

Your objective in studying other sources should be to understand the reasoning behind their approach, and to then work from that understanding. If you are reading from a source and writing your solution at the same time, even with embellishments, that likely constitutes academic misconduct.

Project Structure

For this project you are given four (4) programming problems to solve, all of which are themed around shipping and logistics operations. Detailed descriptions are given in the Problem Specifications section, but as a brief summary:

`Cargo` : Keep track of what total mass of cargo a ship is carrying at points along its route.

`Fines` : Compute the total number of fines issued for ships failing to give way.

`Shallows` : Find the maximum draught your ship can have while being able to reach other ports.

`Subsidiaries` : Determine whether a payment is internal to a company, and if so, which.

Each problem is given in a separate directory, and can and should be considered independently from the other three problems.

Problem Structure

Each problem is given in its own subdirectory, which follow a common structure. In this section we will use the `Cargo` problem as an example. You should have received the following directory structure:

```
Cargo/
├── Cargo.java
├── CargoImpl.java
├── CargoTester.java
└── testdata
    ├── 00_small/...
    └── 01_large/...
```

`Cargo.java` : Java interface file. Do not modify this file. Your solution (`CargoImpl.java`) must implement this interface.

`CargoImpl.java` : Template Java class file. Modify this file to implement your solution to the problem.

`CargoTester.java` : Testing utility. Compile and run to automatically test your code against the provided `testdata`.

`testdata` : Directory containing test cases used by `CargoTester`.

- `00_small` contains the "small" test cases.
- `01_large` contains the "large" test cases.

Implementation

You are expected to implement a solution to each problem.

When implementing your solution you should only need to modify `CargoImpl.java`. You should not create or modify any other code files.

You may import anything you desire from `java.util` or `java.lang`. You may ***not*** import anything from any other sources, including the `CITS2200` package.

Compiling and Testing

Once you have implemented your solution in `CargoImpl.java`, you can compile it and the test utility with:

```
javac Cargo.java CargoImpl.java CargoTester.java
```

If this does not successfully compile, then there is an error with your code, if not, you can then run the tests using:

```
java CargoTester
```

You should ideally see output something like:

```
Testing Cargo:
    00_small:
        00_empty: PASS
        01_steps: PASS
        ...
    All tests passed
    01_large:
        ...
    All tests passed
All tests passed
```

The tester will report the status of each test case:

`PASS` : Your solution ran and gave the correct answer

`FAIL` : Your solution ran but gave an incorrect answer

`ERROR` : Your solution encountered an error and gave no answer

`TIME_LIMIT_EXCEEDED` : Your solution did not give an answer within the two (2) second time limit

`UNKNOWN` : The test was not run

By default the tester stops after the first test that does not pass. You can force it to run all tests regardless with `java CargoTester all`.

Explanation

You must provide a written explanation of why your solution will give the correct answer, and of the time complexity of your solution.

Note that there is no need to describe the step by step process of your algorithm, as the code already does that. Instead you should aim to write a logical argument sufficient to convince a fellow student (or a marker) that your solution will always give the correct answer.

Submission

You must submit exactly two files (for each problem):

`CargoExplanation.pdf` : A **PDF** of your explanation of your chosen solution

`CargoImpl.java` : The Java source code of your completed implementation of your chosen solution

Across all four problems you are therefore expected to submit eight files:

- `CargoExplanation.pdf`
- `CargoImpl.java`
- `FinesExplanation.pdf`
- `FinesImpl.java`
- `ShallowsExplanation.pdf`
- `ShallowsImpl.java`
- `SubsidiariesExplanation.pdf`
- `SubsidiariesImpl.java`

Every file you submit as part of your project must include your full name(s) and student number(s). For PDFs, your name(s) and student numbers(s) should appear somewhere on the first page. For `.java` code files, the first line of the file should be of the form:

```
// Full Name (StudentNum), Full Name (StudentNum)
```

For example, an individual submission may look like

```
// Ada Lovelace (21234567)
```

and a pair pubmission may look like

```
// Ada Lovelace (21234567), Charles Babbage (22345678)
```

If your files do not include your name(s) and student number(s), we may not be able to mark your project, as we may not know who submitted it.

If you are working in a pair, only one of you should make a submission.

Marking

Each problem is marked out of a total of 10 possible marks allocated according to the following rubric. Learning outcomes are as per the Unit Details.

Criterion	Basic	Proficient	Advanced	Total	Outcomes
Coding	(+1) Compiles and runs			/1	3
Correctness	(+4) Passes almost all small tests	(+2) Passes all small test cases		/6	1, 3, 4
Efficiency			(+1) Passes all large tests	/1	5, 6
Explanation		(+1) Some minor errors or omissions	(+1) Thorough and convincing explanation	/2	2, 5, 6

Note that since execution time may vary slightly between runs and between computers, code that passes in one case may exceed the time limit in another. If in doubt, your code must *reliably* run within the time limit on a CS lab computer to receive the marks.

Problem Specifications

`Cargo`

It is important for a cargo ship crew to be able to predict what total mass of cargo they will have on board at points throughout their voyage. You are tasked with writing a system to keep track of the mass of cargo on board at each point along the route as new jobs come in to transport some cargo from one port to another.

Consider some planned route `p[0], p[1], p[2], ...` where each `p[i]` is a port visited along the route. We would like to be able to perform a few different operations:

- Add a job carrying some cargo from one port to another (receiving port can not be before sending port)
- Remove a previously added job
- Find what total mass of cargo we will have at some point along the route

We represent all three of these operations with the `Query` class provided in the `Cargo` interface:

- `cargoMass` is the mass of cargo being transported (may be negative to delete a previous job)
- `collect` is the index of the port at which we will collect the cargo
- `deliver` is the index of the port to which the cargo must be delivered (`collect <= deliver`)

The result of a query is the total mass of cargo expected to be on board when leaving `p[collect]` (after any change caused by the query). The effects of each query are persistent and will impact the results of later queries.

For example:

- `{cargoMass = 4, collect = 2, deliver = 5}` represents adding a job carrying 4 mass units of cargo from `p[2]` to `p[5]`
- `{cargoMass = -4, collect = 2, deliver = 5}` represents deleting the above job

- `{cargoMass = 0, collect = 3, deliver = 3}` represents a query to find the total mass of cargo that will be on board when departing from `p[3]`

Implement a function `int[] departureMasses(int stops, Query[] queries)`, where `stops` is the number of stops on the route and `queries` is a sequence of queries. The function should perform the query operations in the order given, and return an array of the result of each query in the same order.

You may assume `0 < stops <= 10^6`, `0 < queries.length <= 10^6`, `0 <= collect <= deliver < stops`, and no total cargo mass at any point will overflow an `int`.

Where `N` is the number of stops and `Q` is the number of queries, a solution with a computational complexity of `O(N Q)` is not expected to pass the large test cases, but most faster algorithms should.

Example

Consider `departureMasses(9, queries)`, with queries of the form `{cargoMass, collect, deliver}` as follow:

- Total cargo mass when leaving each of ports `p[0]` through `p[7]` is initially `{0, 0, 0, 0, 0, 0, 0, 0}` (note we never leave `p[8]`)
- `{4, 2, 5}` gives `{0, 0, 4, 4, 4, 0, 0, 0}`, result is `4`
- `{3, 3, 8}` gives `{0, 0, 4, 7, 7, 3, 3, 3}`, result is `7`
- `{0, 4, 4}` gives `{0, 0, 4, 7, 7, 3, 3, 3}`, result is `7`
- `{-4, 2, 5}` gives `{0, 0, 0, 3, 3, 3, 3, 3}`, result is `0`

Therefore `departureMasses` would return the array `{4, 7, 7, 0}`.

`Fines`

It is not uncommon to find choke points or "narrows" in harbours around the world. In busy harbours, a harbourmaster will assign ships a priority. Ships with lower priority must gives way to allow ships with higher priority to pass through the narrows. If a ship fails to give way, the harbourmaster may issue that ship a fine for each higher priority ship it cut off. Note that multiple ships may be given the same priority, and ships with the same priority are not required to give way to each other.

Implement a function `long countFines(int[] priorities)` that counts the total number of fines the harbourmaster may issue. The array `priorities` is a list of the priority of each ship that passed through the narrows in the order they passed through.

Where `N` is the number of ships that passed through the narrows, a solution with a computational complexity of `O(N^2)` is not expected to pass the large test cases, but most faster algorithms should.

Example

Consider `countFines({ 4, 9, 7, 2, 1, 3 })`:

- The ship with priority 4 failed to give way to two other ships (9 and 7), 2 fines
- The ship with priority 9 did not fail to give way
- The ship with priority 7 did not fail to give way
- The ship with priority 2 failed to give way to one other ship (3), 1 fine
- The ship with priority 1 failed to give way to one other ship (3), 1 fine
- The ship with priority 3 did not fail to give way

Therefore `countFines` should return the total of `4` fines.

`Shallows`

Most of the time when ships travel between ports it is along shipping lanes. Knowing various properties of these shipping lanes helps avoid accidents. One such property is the depth of the shallowest point along the lane (that is, the minimum depth of the lane).

The vertical distance between the surface of the water and the bottom of the ship is called the draught. A ship with a draught greater than the depth at the shallowest point of a lane is unable to use that lane, as it would run aground, risking getting stuck or causing serious damage.

For some specified origin port, we would like to know the maximum draught a ship can have while still being able to safely make it to some other port. Note that a ship does not have to travel directly to its destination port, but can use any sequence of lanes and visit other ports along the way in order to avoid shallow lanes.

Implement a function `int[] maximumDraughts(int ports, Lane[] lanes, int origin)` that computes the maximum draught of ship that can safely travel from the specified origin port to each other port.

- `ports` is the number of ports
- `lanes` is a list of shipping lanes
  - `Lane.depart` is the index of the port from which the shipping lane departs
  - `Lane.arrive` is the index of the port at which the shipping lane arrives
  - `Lane.depth` is the depth of the shallowest point along the shipping lane
- `origin` is the index of the origin port

The return value should be an array `d` such that `d[i]` is the maximum draught a ship can have while still being able to reach port `i` (even indirectly). Since we start at the origin, and so do not need to use any lanes to reach the origin, `d[origin]` should be `Integer.MAX_VALUE`. If there is no route from `origin` to `i` by any means, `d[i]` should be 0.

You may assume `ports > 0`, `0 <= origin < ports`, `0 <= Lane.depart < ports`, `0 <= Lane.arrive < ports`, and `Lane.depth >= 0`.

Where `P` is the number of ports and `L` is the number of lanes, a solution with computational complexity of `O(P^2)` or `O(P L)` is not expected to pass the large test cases, but most faster algorithms should.

Example

Consider `maximumDraughts(5, lanes, 0)` with lanes of the form `{depart, arrive, depth}` as follow:

- `{0, 1, 9}`: Lane from port 0 to port 1 with shallow point 9 units deep
- `{0, 2, 2}`: Lane from port 0 to port 2 with shallow point 2 units deep
- `{0, 3, 1}`: Lane from port 0 to port 3 with shallow point 1 units deep
- `{1, 2, 7}`: Lane from port 1 to port 2 with shallow point 7 units deep
- `{1, 3, 2}`: Lane from port 1 to port 3 with shallow point 2 units deep
- `{2, 3, 8}`: Lane from port 2 to port 3 with shallow point 8 units deep
- `{4, 2, 9}`: Lane from port 4 to port 2 with shallow point 9 units deep

The safest routes from the origin to each vertex are:

1. `0-1`: Shallowest lane = 9
2. `0-1, 1-2`: Shallowest lane = 7
3. `0-1, 1-2, 2-3`: Shallowest lane = 7

4. There is no route to port 4

Therefore `maximumDraughts` should return the array `{Integer.MAX_VALUE, 9, 7, 7, 0}`.

`Subsidiaries`

Many shipping and logistics companies are in fact wholly owned subsidiaries of other shipping companies. This becomes particularly complicated when these companies start making payments to each other.

Of course these companies are interested in "avoiding" paying tax wherever possible, so for any given payment, if they can find a company that (directly or indirectly) owns both companies involved in the transaction, then they can claim the payment is internal to that company and not have to pay tax on it. They don't want the government to catch on, though, so they would prefer to use the smallest company possible. For these purposes the size of a company is how many other companies it owns (directly or indirectly), so a company is always considered larger than any of its subsidiaries.

Implement a function `int[] sharedOwners(int[] owners, Query[] queries)` that finds for each transaction query the smallest company for which the payment could be considered internal, if there is one.

- `owners` is an array such that `owners[id]` is the id of the company that owns `id`, or `-1` if `id` is not a subsidiary of any other company
- `queries` is an array of `Subsidiaires.Query`s corresponding to each transaction
  - `Query.payer` is the id of the paying company
  - `Query.payee` is the id of the company being paid

The return value should be an array `s` such that `s[i]` is the id of the smallest company that (directly or indirectly) owns both the `payer` and the `payee` from `queries[i]`, or `-1` if no such company exists.

You may assume `-1 <= owners[i] < owners.length`, `0 <= payer < owners.length`, `0 <= payee < owners.length`.

Where `N` is the number of companies and `Q` is the number of queries, a solution with computational complexity of `O(N Q)` is not expected to pass the large test cases, but most faster algorithms should.

Example

Consider `sharedOwners({-1, 0, 0, -1, 3}, queries)` with queries of the form `{payer, payee}` as follow:

- `{0, 2}`: 0 owns 2, so the result is 0
- `{1, 2}`: 0 owns both 1 and 2, so the result is 0
- `{1, 4}`: There is no company that owns both 1 and 4, so -1
- `{3, 4}`: 3 owns 4, so the result is 3

Therefore `sharedOwners` sould return the array `{0, 0, -1, 3}`.