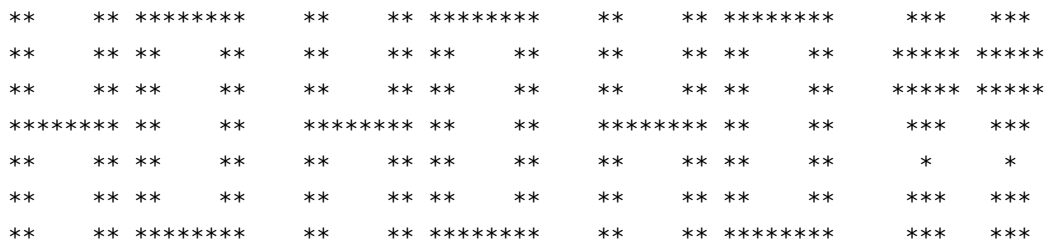
Java Practice Test

Thursday 15th December 2016 14:00 – 17:00

THREE HOURS

(including 10 minutes planning time)



- Please make your swipe card visible on your desk.
- After the planning time, log in using your username as **both** your username and password.

The maximum total is 25.

Credit will be awarded throughout for clarity, conciseness, useful commenting, and appropriate use of assertions.

Important note: THREE MARKS will be deducted from solutions that do not compile. You should comment out any code which you cannot get to compile.

The Travelling Santa Problem

The Travelling Santa Problem (TSP – which many computer scientists erroneously mistake to mean *Travelling Salesman Problem*) is a problem that Santa faces every Christmas. With many presents to deliver to different towns and villages, Santa would like to reduce the time spent travelling as much as possible. Given a starting location and a number of places on a map that he is planning to visit, the problem is to find the optimal (i.e. shortest) route, visiting each town exactly once, and at the end returning to the starting location.

This is generally known as a hard problem to solve computationally, as the perfect route cannot be computed in polynomial time. For `n` towns, including the starting location, there are `n-1` options for the first stop, `n-2` for the second stop, and so on. In other words, there are `O(n!)` possible routes, making it hard to find the optimal solution in a reasonable amount of time.

Many heuristic approaches have been tried to find an approximate solution. One particularly straight forward method is the *nearest-neighbour* algorithm. The idea is that Santa should always choose to travel to the nearest place that has not yet been visited, until eventually all places are visited and Santa has returned back to his starting location. While this may not result in the optimal route, it has nevertheless been shown that on a randomised map with a uniformly distributed set of cities, the nearest-neighbour route is only around 25% longer than the most optimal route on average. In practice, there are also specific scenarios where the nearest-neighbour algorithm yields the worst possible route, but in many cases, it yields a reasonable and simple solution.

In this test, you will be implementing the nearest-neighbour algorithm, to help Santa plan a route through the towns he is planning to visit, and afterwards return back to his home. You will initially be provided with a map in form of a string, which needs to be parsed into an internal representation. The map contains details of where the locations are which Santa will need to visit during his Christmas travels. After determining a route, the map can be reprinted, with towns numbered in the order in which they will be visited.

Getting Started

In the `TravellingSantaProblem` directory in your Lexis home directory you will find nine `.java` files.

Five of these files you should not need to edit, but will need to make use of during this exercise. They provide:

- `Coordinate.java`: contains a class `Coordinate` which stores two `int`s, representing an x, y location on the map. It provides an `int getX()` and `int getY()` to access the `x` and `y` coordinates, respectively, an `int distanceTo(Coordinate c)` method to calculate the distance to another coordinate `c`, and a `String toString()` method. (See below for a discussion on distances).
- `MapEntry.java`: contains an enum `MapEntry` which represents the eight different types of map entries. The full list can be found in the description of Part I a, below.
- `Maps.java`: contains three maps of different sizes.
- `Tests.java`: contains a number of tests. As the tests use assertions, be sure you execute your code with '`-ea`', to enable assertion checking.
- `TravellingSantaProblem.java`: provides a main method to run the Travelling Santa Problem on the three maps provided.

The remaining files you should edit, filling in the stub methods as explained below. Once these are completed you should be able to run the Travelling Santa Problem and see the route marked out on the map.

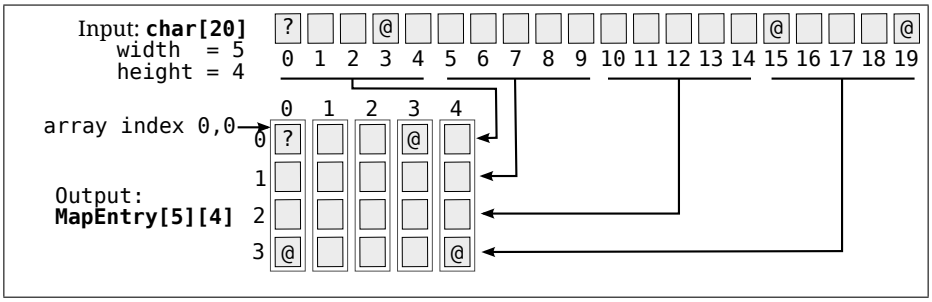
- `MappingUtils.java`: contains stubs for static utility methods which you will need to implement as part of this exercise.
- `SantasMap.java`: contains the class `SantasMap` – that is, the map that Santa holds. It stores a two-dimensional array of `MapEntry` for the map, and an array of `Coordinate`, which contains coordinates for the stops Santa will have to make on his trip. Method stubs are provided which, when completed, will be able to parse the stops from the map, and compute a distance matrix, which will be needed for the nearest-neighbour algorithm. It also provides a method to help you sort the array of coordinates, which will help with testing, as well as `MapEntry[][] getMap()` and `Coordinate[] getStops()` methods.
- `Route.java`: contains the class `Route`, which is used to store the route Santa will travel. You are provided with method stubs which, when completed, will add a place to the current route, compute the total length of the route, and annotate `SantasMap` with the route – `void printAnnotatedMap()` method provided for you.
- `RouteUtils.java`: contains stubs for static utility methods to determine the total number of possible routes and to find the route Santa should travel using the nearest-neighbour algorithm.

You should not change the signatures of any of the provided methods: auto-testing of your solution depends on the methods having exactly their original signatures. However you may add additional methods and classes (e.g. for testing) as you see fit. Any new Java files should be placed in the `TravellingSantaProblem` directory.

To help you test your work as you go, we have provided a test suite for all of Part I and Part II in `Tests.java`. The tests are checked using `assert`, so only the first failure will be reported. If you attempt the questions in a different order you may need to comment out or re-order the method calls in the `main` method of this file.

Working with maps

Maps are represented as a string of ASCII characters. See the following diagram for an example input and output. Notice that in the output array index $(0, 0)$ is the *top left* corner of the board. This means the first character in the input array ends up in the first row of the output array. Note, that x-coordinates reference columns, while y-coordinates reference rows.



The map marks Santa's starting location, denoted by a '?', as well as three towns, marked as '@'. From this, we can construct a list of places on this map, sorted first by their y-coordinate and then by their x-coordinate, ascendingly. The following table shows a sorted list of places:

place	coordinate
A	`out[0][0]`
B	`out[3][0]`
C	`out[0][3]`
D	`out[4][3]`

In order to find a route and evaluate its cost, it is necessary to establish a concept of distances. You will find that the `Coordinate` class provides code to handle this. Santa travels 'as the reindeer sleigh flies', so the formula to compute distances is straight-forward. As character fields are not square and and their height is approximately equal to twice their width, we say that one field is 1km wide and 2km high, in order to deal with implicit skewing on a printed character map. From this, we can compute a distance matrix (in metres), which will be the basis for the nearest-neighbour algorithm:

	A	**B**	**C**	**D**
A	0	3000	6000	7211
B	3000	0	6708	6082
C	6000	6708	0	4000
D	7211	6082	4000	0

What to do

Part I: MappingUtils.java (total: 7 marks)

This class contains some static utility methods that are concerned with parsing and outputting maps, which can be used elsewhere in the program. You will need to complete these to make implementing the rest of the route planning easier.

a. MappingUtils.java (5 marks)

The descriptions of maps are provided in an array of `char`s. Each ASCII character in the array corresponds to one entry, as detailed in the following table:

`MapEntry`	`char` form
`EMPTY`	`' '`
`FOREST`	`'*'`
`WATER`	`'~'`
`MOUNTAINS`	`'^'`
`LAND_BORDER`	`'#'`
`VILLAGE`	`'.'`
`TOWN`	`'@'`
`SANTAS_STARTING_LOCATION`	`'?'`

1. Implement a static method `MapEntry charToMapEntry(char c)` – which converts an ASCII character into the corresponding `MapEntry`.

   You do *not* need to create separately named constants for each of the ASCII characters in this method.

2. Implement a static method `char mapEntryToChar(MapEntry e)` – which converts a `MapEntry` back to its corresponding ASCII character.

3. Hence, implement a static method
   `MapEntry[][] parseMap(char[] description, int width, int height)`
   – which, given the ASCII representation of a map and its associated width and height,

this method should assume `description` is `width`×`height` in length), builds a rectangular two-dimensional array of `MapEntry`. The outer array should have length `width`, and the inner arrays should each have length `height`.

4. Implement a static method `boolean isStop(MapEntry place)` – which returns whether the given `MapEntry` will be a stop on Santa's route. Santa will have to stop at each village and town, and later on return to his starting location, which will mark the first and last stop on the route.

b. Finding the total number of possible routes (2 marks)

We wish to know how many possible routes Santa could take through a set of towns. Starting from a given place, each town or village must be visited exactly once, before returning to the starting location. For n places including the starting location, the solution is $(n-1)!/2$. As routes symmetric and can be travelled in either direction at the same cost, the total number of routes may be divided by two. Santa is interested in establishing the total number of routes from his starting location through n cities (i.e. where n does *not include* his starting location).

In `RouteUtils.java`, implement a static method `long getNumRoutes(long stops)` as described. Beware of edge cases. You do not have to handle overflows.

Part II: SantasMap.java (total: 6 marks)

The `SantasMap` class is initialised with a two-dimensional array of `MapEntry`, and when fully implemented will provide functionality that will form the basis of the nearest-neighbour algorithm, and for outputting annotated maps outlining the route Santa will ultimately travel. The class provides a static final field `MAX_STOPS` which records the maximum number of stops you will need to be able to handle in this exercise.

1. Implement a private instance method `Coordinate[] findStops()`, which finds a unique set of locations of all places where Santa will have to travel to on his route, including his starting location. An array of the correct size should be returned. This function acts as a helper function to `Coordinate[] findStopsSorted()`, which has been provided for you and sorts the result of your function, first by their `Y`-coordinate, then by their `X`-coordinate. Results of your function are visible only by calling `Coordinate[] getStops()`, and its visibility should not be changed.
2. Implement a public instance method `int[][] getDistanceMatrix()`, which returns a matrix recording the distances between all pairs of stops. The rows and columns of the matrix should be in the same order as the coordinates returned by `Coordinate[] getStops()`, such that entries in the distance matrix can be mapped to coordinates. As Santa travels by reindeer sleigh and does not rely on roads, the matrix will be made up of straight-line distances. As distances are equal in either direction of travel, the matrix will be symmetric, and Santa has asked that the matrix be computed with the least amount of computation possible. Consider the example of the distance matrix above: Only half the values need to be computed due to its symmetric properties, while the other half of values can be copied.

Part III: Route.java (total: 4 marks)

The `Route` class is used to keep track of the route Santa wishes to travel. The route is stored on the class as a field `private final List<Coordinate> stops`. The first and last stop of any route should be Santa's starting location.

1. Implement a public instance function `void add(Coordinate place)`, which adds a coordinate to the current route.
2. Implement a public instance function `int totalLength()`, which returns the total length of the route.

Part IV: Nearest-Neighbour Routing Algorithm (total: 6 marks)

In `RouteUtils.java`, implement a static function `Route findNearestNeighbourRoute(SantasMap map)` – which, given a `SantasMap`, returns a `Route`, using the nearest-neighbour algorithm. The algorithm should work as follows:

1. In a sorted list of stops as stored inside `SantasMap`, Santa's starting location will always be the first one. (This is due to the fact that Santa's starting location will always be the northern-most stop on any of the maps provided).
2. Santa thinks it may prove helpful to have a check-list to keep track of which places have been visited and which have not, as he does not want to visit any place more than once on his trip.
3. From each location, Santa wishes to travel to the nearest place that has not yet been visited, which he would like to identify and add to his `Route`. (If two places are equally near and neither have been travelled to, he will travel to the one that comes first in his list – i.e. that has the lower index in the `Coordinate` array).
4. After all places have been visited, Santa returns to his starting location.

Part V: Annotating Maps (total: 2 marks)

Finally, the obtained `Route` should be printed on an annotated map. In `Route.java`, implement an instance function `char[][] getAnnotatedMap()`, which translates Santa's map into text form, as a two-dimensional array of characters. Characters for towns and villages should be replaced by numbers, such that Santa's starting location is numbered `0`, subsequent towns and villages are numbered first `1-9`, and for any further places the uppercase characters `A-Z` are used instead of numbers. See the Figure 1 for an example.

```
?*****
****@*
**@**@
@***@*
```

(a) Original map

```
0*****
****1*
**4**2
5***3*
```

(b) Annotated map

Figure 1: A small map with five cities showing the order in which they will be visited when applying the nearest-neighbour algorithm

Once you are done, run the `main` method in `TravellingSantaProblem`, to watch the results.

Total across all parts: 25 marks