Data Structures

Overview

When you go Trick-or-Treating, you'll encounter many neighborhoods, houses, candy baskets, and types of candy. We can represent the network formed by these neighborhoods, houses, baskets, and candies as a weighted undirected graph. This allows us to determine the best ways for children to collect the most treats, find the shortest routes, and see if they can finish their route in time to avoid curfew.

This is an undirected graph assignment where vertices are houses. An edge represents a sidewalk between two houses or crossing a street.

Implementation

Overview of Files

- We provide one Java class for each of the tasks you need to complete. Update these files with your code for each task (see Implementation Notes).
- We provide StdIn and StdOut.
- We provide a set of input files to represent a graph. You are free to edit them, as well as create new input files with the correct format to help test your code.

Implementation Notes

The structure of this assignment is quite different from the previous assignments. Please play careful attention to the following rules.

- **DO NOT** use static variables in your code.
- In each given Java class, you will read from a given input file (passed in as command line arguments), read some other information (like source vertices, etc) and write to a given output file (whose name is passed in as a command line argument).
- **DO NOT** change the names of any of the given Java files, or the project structure itself (do not change directory names or create new directories).
- **DO NOT** remove the package statement from any of the given input files.
- **DO NOT** use System.exit() in your code.
- Unlike any previous assignment, **YOU MAY** (and should) create your own classes in the src/trick folder. **YOU MAY** import "java.util.*", but **DO NOT** import anything else. Make sure any new classes have a package statement: package trick;
- The classes that you create **MAY NOT** have spaces in their names.
- In order to grade a problem, we run the corresponding Java class and verify the output file. This means you have full freedom in your project structure, as long as our provided classes output the correct answer to the correct output file. Take this opportunity to practice your project design skills, and write clean code that avoids redundancy.
- **DO NOT** remove the package statement from the provided classes.

Using StdIn and StdOut

- Use **StdIn**.setFile(fileName) to set the current input file you want to read from.
- You can now use methods like **StdIn**.readInt(), **StdIn**.readString() and **StdIn**.readLine() to operate on the input file as if it was standard input.
- The methods **StdIn**.readInt() and **StdIn**.readString() actually leave the newline character unread, so if you use **StdIn**.readLine() after one of these methods, it will read this character rather than the next line. If you want to read the next line with **StdIn**.readLine(), you will need to call **StdIn**.readLine() once to read the newline character and then again to read the next line. **StdIn**.readInt() and **StdIn**.readString() ignore spaces and newlines by default.
- Use **StdOut**.setFile(fileName) to set the current output file you want to write to. It creates the file if it doesn't already exist.
- You can now use methods like **StdOut**.print() and **StdOut**.println() to operate on the output file as if it was standard output.

Autolab **ignores** empty lines and extra spaces that your output files may have.

Tasks to be implemented by you:

Mini-Tasks: Create a Graph (strongly recommended)

This section is meant to guide you towards implementing the graph data structure you'll need to work with for the rest of this assignment. These mini-tasks are things you'll have to do in the first assignment task, but we are providing these mini-tasks to guide you through some of the steps in NeighborhoodMap, and come up with a streamlined graph implementation for the rest of the assignment.

This is an undirected graph assignment where vertices represent houses. An edge represents a sidewalk between two houses or crossing a street.

Mini-Task 1: House class

We can create a class that stores a house name and weight (we'll handle candies separately a bit later; those will be stored outside a House object). If you're using a linked list implementation for your adjacency list, you should also have a next reference that points to the next House.

When the edge weight is 0, the House will represent a **vertex**. When the edge weight is greater than 0, the House will represent an **edge** and would be associated with a "from" house via our adjacency list.

Mini-Task 2: Graph class

You'll need to create an adjacency list to store edges corresponding to a vertex: we recommend using the built-in classes in the java.util package to do this.

- You can use the House class to store vertices and edges. Recall that a House whose weight is 0 represents a vertex, and a House whose weight is > 0 represents an edge.

Possible approaches:

1. an adjacency list (array of house lists), where the front of each list corresponds to a vertex and its successive nodes represent edges incident to that vertex. More formally, we use an array of lists where adj[i] refers to a vertex and its next nodes refer to edges of that vertex. This is similar to the lab.

Graph class:

2. OR: You can also use a HashMap where keys are house names and values are the list of edges.

Graph class:

keys: vertices

h1	h2

values: list of houses (represent edg

How do HashMaps work?

- HashMaps store (key, value) pairs, and allow you to look up a value for a corresponding key.
- Initialize a HashMap of key type Key and value type Value using HashMap<Key, Value> hashMap = new HashMap<>();
- Put a new value (or update an existing value) for a key using hashMap.put(key, newValue);
- Use hashMap.putIfAbsent(key, value) to put a key, value pair in the HashMap if a key doesn't exist.
- You can check if the HashMap contains some key in average case O(1) time (returns a boolean) with name.containsKey(key)
- You can check the value of some key in the HashMap in average case O(1) time with name.get(key)
- You can iterate over all the keys in the HashMap with for (Key key : name.keySet()) where Key is the type of keys in the HashMap.

Mini-Task 2.1: addEdge

Create a method, in the Graph Class, to insert an edge into your graph. This will insert a single directed edge to the graph.

If the vertex doesn't exist in your adjacency list, add the edge as the first edge associated with this vertex. Otherwise, add the edge to the **end** of the vertex's corresponding list.

Be sure to set the weight correctly.

Mini-Task 2.1: adj(v)

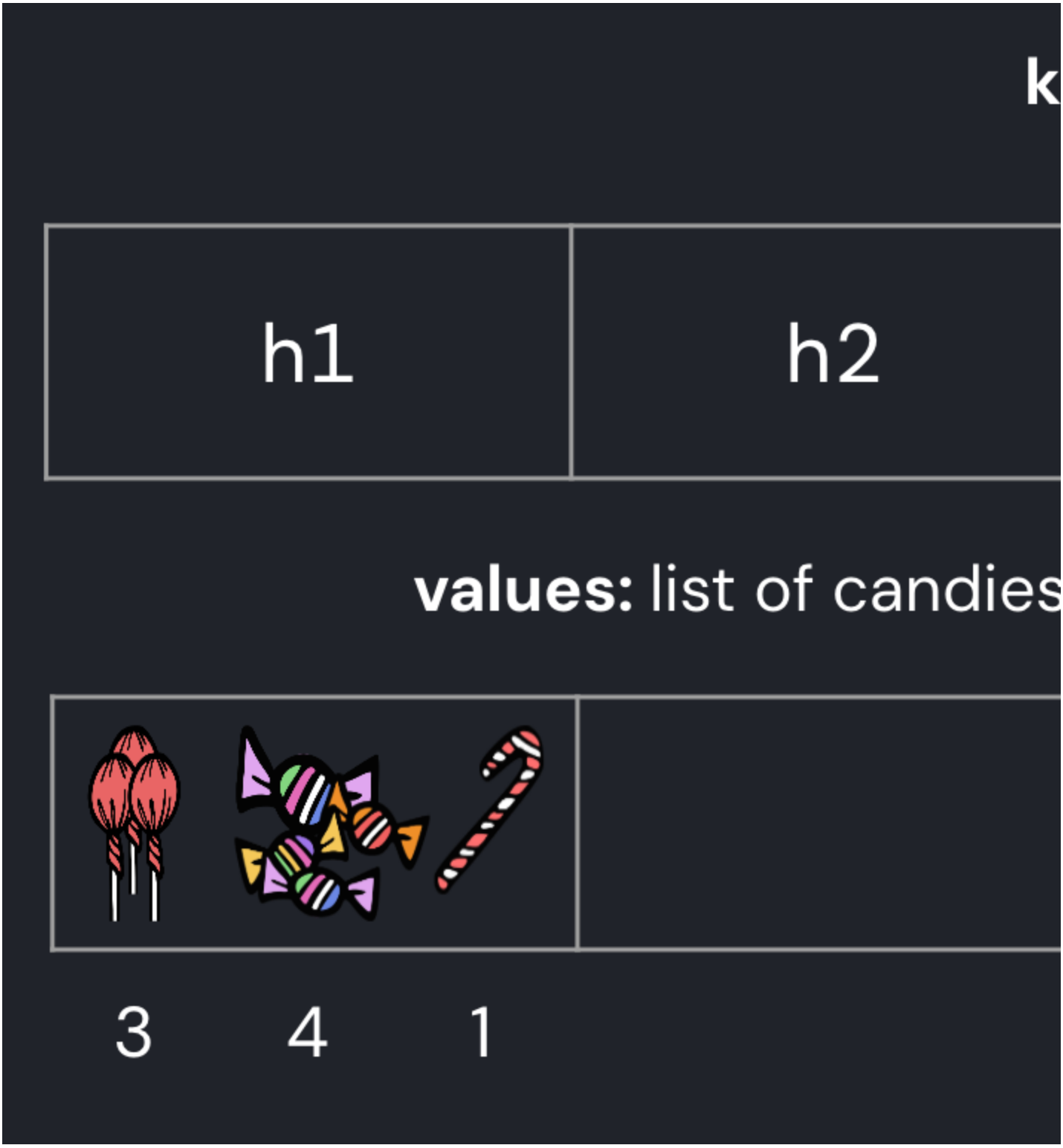
Create a method, in the Graph class, to return the edges adjacent to a given vertex. This method is similar to the adj method discussed in class. You'll need this in later tasks beyond NeighborhoodMap.

Mini-Task 3: Storing Houses + Candies

To refer to houses more efficiently, we'll need to find a way to map a house name to the rest of the house's information (like its inventory). In the adjacency list, a vertex has its own list of incident edges.

You may want to use a HashMap of (house name, list of candies). To get a list of candies for a given house name, you can use **hashMap.get(houseName);**

- Recall that candies consist of a name and count. You can either **have a list of Candy objects** that contain a name and count, or use a HashMap as your list of candies, storing another set of key-value pairs: (candy name, candy count)
- This will allow you to work with only house names in your adjacency list, with a way to look up candy inventories for a house.



These mini-tasks will prepare you for the main tasks, as you'll have a graph implementation you can work with for each task.

NeighborhoodMap.java

In this task, you'll initialize the graph data structure that you'll use for all later tasks.

If you completed the mini-tasks, you will use the Graph class to store the graph from the input file and you'll be able to work with the other classes you've created.

Create the neighborhood map using the input file passed through command line arguments. In NeighborhoodMap.java, you will read the input file name from args[0] and output to args[1]. The input file is formatted as follows:

- The number of houses, say n
- n lines containing each house (a string), the number of candies, and for each candy its name and quantity (all space-separated)
- The number of edges, say e
- e lines containing the first vertex (a house/string), the second vertex (a house/string), and the corresponding edge weight (as an integer)

Use the `StdIn` library to read from a file:

- **`StdIn.setFile(filename)`** opens a file to be read.
- **`StdIn.readInt()`** reads the next integer value from the opened file (whether the value is in the current line or in the next line).
- **`StdIn.readString()`** reads the next String value from the opened file.

Format your output as follows:

- For each house, output the house name followed by candies (name and quantity, space separated) on the same line.
- Then, print the adjacency list. There should be n lines (n = number of houses) containing the house name followed by, for each edge, the other house name and its edge weight (space separated).

Compile command: **javac -d bin src/trick/*.java**

Run command: **java -cp bin trick.NeighborhoodMap input1.in input1.out**

Below is the expected output for NeighborhoodMap.java when running with arguments “input1.in input1.out”. Order of lines does not matter as long as candy lines come first and the order of edges for each vertex matches. The order of candies doesn’t matter.

```
h8 KitKat 4
h9 KitKat 3 Skittles 2
h1 KitKat 5 Skittles 5
h2 KitKat 4
h3 Skittles 3
h4 KitKat 8 Skittles 5
h5 KitKat 0
h6 KitKat 0
h7 KitKat 7
h8 h9 18 h5 20
h9 h6 2 h8 18
h1 h4 15 h2 4
h2 h3 2 h7 17 h1 4
h3 h2 2 h7 3 h4 11 h6 10
h4 h1 15 h6 10 h3 11 h5 14
h5 h8 20 h7 8 h4 14
h6 h4 10 h9 2 h3 10
h7 h3 3 h5 8 h2 17
```

Submit the entire TrickOrTreat directory with NeighborhoodMap.java implemented under Early

Submission to receive extra credit (see "Zipping the directory for submission" below).

DO NOT submit only NeighborhoodMap.java to Autolab.

FindTreatsRoute.java

Let's try to find all houses reachable from some other house: we can use DFS to do this, as it backs out when all vertices reachable from some source have been visited. Given a source vertex, produce a DFS (depth-first search) ordering of **all** vertices reachable from the source, in the order given by DFS.

Print out one line with all houses in the order they were visited, space-separated.

args[0] is the neighborhood map input file, args[1] is the source house name, and args[2] is the output file.

Compile command: javac -d bin src/trick/*.java

Run command: java -cp bin trick.FindTreatsRoute input1.in h1 findtreats1.out

Below is the expected output for FindTreatsRoute.java when running with arguments "input1.in h1 findtreats1.out". Order does matter, but extra whitespace will be ignored.

```
h1 h4 h6 h9 h8 h5 h7 h3 h2
```

FindHouseWithMostCandy.java

Now that we know which houses we can reach, let's try to find the house where we can get the most of a specific candy from. We'll use the DFS ordering from before, so you should:

- Traverse each house from the DFS ordering you obtain from FindTreatsRoute.
- Check each house and find the house with the maximal amount of your target candy.

Print out the house name with the most of that specific candy, and only the house name.

args[0] is the neighborhood map input file, args[1] is the source house, args[2] is the candy you're looking for, and args[3] is the output file.

Compile command: **javac -d bin src/trick/*.java**

Run command: **java -cp bin trick.FindHouseWithMostCandy input1.in h1 Skittles findcandy1.out**

Below is the expected output for FindHouseWithMostCandy.java when running with arguments "input1.in h1 Skittles findcandy1.out". Order does matter, but extra whitespace will be ignored.

```
h6
```

PathToMostCandy.java

Let's try to find a path to the house with the most candy. We can use BFS to find the quickest path (with respect to number of hops) from a source house to every other house. Your task is to print out the BFS path from the source house to the house with the most of a specific candy.

- USE the house from FindHouseWithMostCandy and find a path to that house.

- Use the edgeTo array to store predecessors of each vertex, as discussed in class. edgeTo[source] should be null, as there is no cycle.
- To get the path, you will follow an algorithm similar to quick-union's `find()` method, chasing up from the target until the source is in your path.
    - Reverse the path so that it starts from the source — you can use `Collections.reverse(___)` to do this if your path is an ArrayList, for instance.

```
v = target house (most candy)
path = []
while v is not null:
    add v to path
    v = edgeTo[parent];
reverse path to start from source
```

Compile command: **javac -d bin src/trick/*.java**

Run command: **java -cp bin trick.PathToMostCandy input1.in h1 KitKat mostcandy1.out**

Below is the expected output for FindHouseWithMostCandy.java when running with arguments "input1.in h1 Skittles mostcandy1.out". Order does matter, but extra whitespace will be ignored.

```
h1 h4 h6
```

ShortestPath.java

You will use Dijkstra's Algorithm to set the shortest route to all houses from the house passed through in args[1]. For each vertex, output its predecessor in the pred set (you should have v lines with the vertex name and its predecessor space-separated).

- We're using Dijkstra's Algorithm to minimize the time it takes to travel from one house to another.
- Dijkstra's AlgorithmLinks to an external site. video and slidesLinks to an external site.

As stated before, parse the input file from args[0] using the same pattern described in NeighboringHouses.

Use the following pseudocode to complete this algorithm:

```
Algorithm:
let s be the source vertex
done = {} Vertices with known optimal path from source
fringe = {} Vertices with unknown optimal path from source
for each vertex other than source:
  d(v) = infinity
  pred(v) = null
d(s) = 0 Optimal distance from source to parameter vertex
pred(s) = null
add s to fringe
while the fringe is not empty fringe = to-do list, not ideal yet
  m = remove minimum distance vertex from the fringe
  add m to done set
  for each neighbor w of m not in the done set
        if d(w) is infinity then
        d(w) = d(m) + w(m,w)
        add w to fringe
        pred(w) = m
        else if d(w) > (d(m) + w(m,w)) then
        d(w) = d(m) + w(m,w); pred(w) = m
        endif
  endfor
endwhile
```

Your output should be formatted as follows:

- h lines where h is the count of all houses reachable from the source vertex, including the source
- for each line, print the house name followed by its predecessor (given by pred(v)), space separated
- **NOTE:** pred(source) is null; we do not have an edge from the source to itself.

Compile command: **javac -d bin src/trick/*.java**

Run command: **java -cp bin trick.ShortestPath input1.in h1 shortestpaths1.out**

Below is the expected output for ShortestPath.java when running with arguments “input1.in h1 shortestpaths1.out”. Order of lines does not matter, but order within lines does.

```
h8 h9
h9 h6
h1 null
h2 h1
h3 h2
h4 h1
h5 h7
h6 h3
h7 h3
```

CanAvoidCurfew.java

We want to be able to go to another house but we also have a curfew after which we can’t be outside. Let’s see if we can make it to a destination before the curfew, as given by its ideal distance.

- USE the pred and d sets you’d get by calling Dijkstra’s Algorithm on the source vertex. You implemented Dijkstra’s Algorithm in ShortestPaths.

- HINT: d(v) gives you the optimal distance from the source to another vertex, v.

Output true if we can make it, false otherwise, then on the same line the distance to the house space-separated. args[0] is the neighborhood map, args[1] is the source, args[2] is the other house, args[3] is the curfew (an integer), and args[4] is the output file.

Compile command: **javac -d bin src/trick/*.java**

Run command: **java -cp bin trick.CanAvoidCurfew input1.in h1 h8 100 shortestpaths1.out**

Below is the expected output for ShortestPathsBeforeCurfew.java when running with arguments "input1.in h1 h8 100 shortestpaths1.out".

Helpful Java Classes

The following are some data structures which are automatically imported with "**java.util.***" that can help make your code cleaner and more efficient. You are free to use (or not use) any of these, as well as any other class under "java.util.*". These data structures do not have every method covered here, just some useful ones for this assignment. You can find more information about how to use these classes online.

ArrayList is an ordered array-like structure with no size limit, as it automatically resizes

- You can initialize an empty ArrayList named "name" which holds objects of type "Type" with ArrayList name = new ArrayList<>();
- For example, an ArrayList of integers named "arrList" is initialized with ArrayList arrList = new ArrayList<>();
- You can add a new element of type "Type" to the end of your ArrayList in average case O(1) time with name.add(newElement);
- You can get the element at some index of your ArrayList in O(1) time with name.get(index);
- You can set some index to some new element in O(1) time with name.set(index, newElement);
- You can check if the ArrayList contains some element (returns a boolean) in O(n) time with name.contains(element)

Queue implements a FIFO structure

- You can initialize an empty Queue named "name" which holds objects of type "Type" with Queue name = new LinkedList<>();
- For example, a Queue of integers named "q" is initialized with Queue q = new LinkedList<>();
- You can add a new element of type "Type" to the back of your Queue in O(1) time with name.add(newElement);
- You can get the element at the front of the Queue with name.peek()
- You can get and delete the element at the front of the queue with name.remove()

HashMap is an unordered data structure which stores and retrieves key value pairs

- You can initialize an empty HashMap named "name" that maps objects of type "Key" to objects of type "Value" with HashMap<Key, Value> name = new HashMap<>();
- For example, a HashMap named "map" which maps strings to integers is initialized with HashMap<String, Integer> map = new HashMap<>();

- You can add a new key value pair, or update an existing key with a new value in average case O(1) time with name.put(key, value);
- You can check if the HashMap contains some key in average case O(1) time (returns a boolean) with name.containsKey(key)
- You can check the value of some key in the HashMap in average case O(1) time with name.get(key)
- You can iterate over all the keys in the HashMap with for (Key key : name.keySet()) where Key is the type of keys in the HashMap.

Advanced: LinkedHashMap is an ORDERED data structure that stores and retrieves key-value pairs. It works similarly to HashMaps but .put orders elements similarly to ArrayLists' .add method (add to the last position).

LinkedList is an ordered data structure

- You can initialize an empty LinkedList named "name" which holds objects of type "Type" with LinkedList name = new LinkedList();
- You can add an element to the end of your linked list with name.add(item); and to the front with name.addFirst(item).
- You can get items using name.get(index), name.getFirst(), or name.getLast(), to get an item at a specific index, first index, or last index respectively.
- name.remove() removes the first element from the list, name.remove(index); removes an item at a specific index.

VSCode Extensions

You can install VSCode extension packs for Java. Take a look at this tutorial. We suggest:

- Extension Pack for Java
- Project Manager for Java
- Debugger for Java

Importing VSCode Project

1. Download the zip file from Autolab Attachments.
2. Unzip the file by double clicking it.
3. Open VSCode
    - Import the folder **TrickOrTreat** to a workspace through **File > Open Folder**

Executing and Debugging

- You can run your program through VSCode or you can use the Terminal to compile and execute. We suggest running through VSCode because it will give you the option to debug.
- How to debug your code (general).
- How to run the debugger on this assignment.
- If you choose the Terminal, from TrickOrTreat directory/folder:
    - to compile: **javac -d bin src/trick/*.java**

Zipping the directory for submission (READ THIS before submitting)

Please read this section carefully; we cannot fix errors for you or remove submissions if you follow these instructions incorrectly or miss something.

- VS Code setup + how to submitLinks to an external site. and slides.Links to an external site.
- Be careful when zipping the directory for submission.
- Autolab is expecting the exact directory organization we provided to you.
- Autolab is expecting the zip file to be named **TrickOrTreat.zip**
- All files that you have written **MUST** be in the **TrickOrTreat/src/trick** directory.

To zip the **TrickOrTreat** directory navigate to the parent directory:

- zip -r TrickOrTreat**.zip** TrickOrTreat

Inspect the zip by listing the files in zip without uncompressing it:

- unzip -l TrickOrTreat.zip

Your zip file **MUST** have the following structure:

```
Archive:  TrickOrTreat.zip
  Length      Date    Time    Name
---------  ---------- -----   ----
        0  11-21-2024 00:34   TrickOrTreat/
      334  11-21-2024 00:33   TrickOrTreat/input1.out
        0  11-22-2024 00:28   TrickOrTreat/bin/
        0  11-22-2024 00:28   TrickOrTreat/bin/trick/
      747  11-22-2024 00:28   TrickOrTreat/bin/trick/PathToMostCand
      738  11-22-2024 00:28   TrickOrTreat/bin/trick/ShortestPath.c
      747  11-22-2024 00:28   TrickOrTreat/bin/trick/FindTreatsRout
      747  11-22-2024 00:28   TrickOrTreat/bin/trick/NeighborhoodMa
      744  11-22-2024 00:28   TrickOrTreat/bin/trick/CanAvoidCurfew
     9613  11-22-2024 00:28   TrickOrTreat/bin/trick/StdIn.class
     4463  11-22-2024 00:28   TrickOrTreat/bin/trick/StdOut.class
      768  11-22-2024 00:28   TrickOrTreat/bin/trick/FindHouseWithM
      192  11-21-2024 00:33   TrickOrTreat/input4.in
      288  11-21-2024 00:33   TrickOrTreat/input1.in
      341  11-21-2024 00:33   TrickOrTreat/input2.in
      261  11-21-2024 00:33   TrickOrTreat/input3.in
        0  11-21-2024 00:34   TrickOrTreat/lib/
   384581  11-21-2024 00:33   TrickOrTreat/lib/junit-4.13.2.jar
    45024  11-21-2024 00:33   TrickOrTreat/lib/hamcrest-core-1.3.ja
        0  11-21-2024 00:55   TrickOrTreat/.vscode/
      161  11-21-2024 00:33   TrickOrTreat/.vscode/settings.json
        0  11-21-2024 00:34   TrickOrTreat/src/
        0  11-21-2024 00:36   TrickOrTreat/src/trick/
     9221  11-21-2024 00:36   TrickOrTreat/src/trick/StdOut.java
      514  11-21-2024 00:48   TrickOrTreat/src/trick/CanAvoidCurfew
      530  11-21-2024 00:35   TrickOrTreat/src/trick/FindHouseWithM
      545  11-21-2024 00:35   TrickOrTreat/src/trick/PathToMostCand
    26940  11-21-2024 00:33   TrickOrTreat/src/trick/StdIn.java
      553  11-21-2024 00:35   TrickOrTreat/src/trick/FindTreatsRout
      568  11-21-2024 00:35   TrickOrTreat/src/trick/ShortestPath.j
      508  11-21-2024 00:35   TrickOrTreat/src/trick/NeighborhoodMa
---------                     -------
   489128                     31 files
```

If your ZIP file structure is incorrect, you will receive a message titled “contact your instructor” as a hint in Autolab. Click the file name in Autolab (netid_version_assignment.zip) to view the archive. You should see TrickOrTreat/src/trick to get to your code. The folder names must match exactly.

Your ZIP file should be called TrickOrTreat.zip, case sensitive. It should not be called trick.zip, TrickOrTreat (2).zip, or any other name besides TrickOrTreat.zip. You should also be sure not to submit ONLY NeighborhoodMap.java or a ZIP that only contains NeighborhoodMap.java to the early submission.

Before submission

Collaboration policy. Read our collaboration policy here.

Submitting the assignment.

You will have to submit a zip file. See previous section on how to zip the directory.

Submit *TrickOrTreat.zip* separately via the web submission system called Autolab. To do this, click the *Assignments* link from the course website; click the *Submit* link for that assignment.

Getting help

If anything is unclear, don't hesitate to drop by office hours or post a question on Piazza.

- Post your questions on Piazza (Canvas -> Piazza)
- Find instructors and head TAs office hours *here*
- Find tutors office hours on Canvas -> Tutoring -> RU CATS
- In addition to office hours we have the CSL in Hill 252, a community space staffed with lab assistants which are undergraduate students further along the CS major to answer questions.

Problem by Kal Pandit and Seth Kelley