Q1

1.1

f(x) = 1

Because it maps every input to the same output (aka the same hash value) which wouldn't evenly distribute the data in a HashTable.

1.2

To avoid the collisions/clustering that occurs with linear probing, use **double hashing** as a new probing strategy. Do this by adding an additional hash function g(key) that is called, so the resulting "total" hash function would be the original function h(key), plus g(key) times the probe number. In other words, the new function would be f(key) = h(key) + i * g(key), where **i** is the **ith** probe into the table.

Another strategy would be to use a **separate chaining** hash table, where each index in the hash table is a 'bucket' that stores all the key/value pairs whose hashes map to that index. This would prevent any changes to the hash function but would take up more space in memory than the double-hashing strategy.

1.3

Use two data structures: an **ArrayList<String>** that represents the slips with the students' names, and a **HashMap<String, HashSet<Integer>>** that maps the names of students to a set of the indices of their slips in the ArrayList.

addSlip() will add the name of the student to the end of the ArrayList, then add the index of the new element to the set of indices that corresponds with the student's name in the HashMap. If the size of the set of indices is equal to 10, an exception will be thrown. This should run in O(1) because adding to the end of an ArrayList takes O(1) time, and finding and adding an index to the set in the HashMap also takes constant time.

withdrawSlips() will get the set of indices associated with the name from the HashMap and, for each index in the set, will swap ArrayList[index] with ArrayList[size - 1], remove ArrayList[size - 1] from the ArrayList, and then update the index of the swapped value in the set in HashMap. Finally, it will remove the name of the student from the HashMap.

This should run in O(1) time because performing the swap and removal from the end of the ArrayList takes constant time, the updating of the index value for the swapped element in HashMap takes constant time (as all operations with a HashMap and HashSet take amortized O(1) time), and doing this for all the indices means this group of functions will be performed at most 10 times, which is still constant. Additionally, the final removal of the name from the HashMap should take O(1) time, which leaves the final sum of operations to be O(1) time.

pickTribute() will use an instance of the Random class to generate a random valid index in the ArrayList, retrieve the student's name stored at that index, then call withdrawSlips() passing in the student's name as the parameter. This should run in O(1) time because generating a random integer, looking up an element at a specific index from an ArrayList, and withdrawSlips() all take O(1) time, so the resulting sum of operations is still constant time.

Q2

2.1

Worst-case would be **O(a)** because to find the median-maximum value of the three LinkedLists for pivot selection takes 3a, and to find the median-minimum value of the three LinkedLists would also take 3a. 3a + 3a = 6a, but because this is big-O we drop the constants, which leaves us with O(a).

Worst-case would be **O(n * a)** because comparing two items takes 2a (a is worst-case lookup in the LinkedList), and a worst-case partition would require reading and comparing every element. 2a * n = 2an, and dropping the constants for big-O we get O(n*a).

Worse-case would be **O(an^2)** because a worst-case run of Quicksort would involve running the pivot selection and partition parts of the algorithm n times. n(an + a) = an^2 + an. Because this is big-O, we only use the highest-degree monomial in the expression, which leaves us with O(an^2).

2.2

Heap Sort

The array is sorting from the right (max to min), which was indicative of a heap sort. Upon tracing through the steps, you can see the maxHeap being built (the maxHeap is complete in step 5). Then you can see it starting to sort from the right as 'nodes' are getting 'removed' from the heap.

2.3

Insertion Sort

Because the "sorted" part of the array starts from the left and grows until the array is fully sorted. So at step 1, only the first element is sorted, and at step 5, the first 5 elements are sorted, so on and so forth.

2.4

Radix Sort

We perform radix sort by looking at the numbers in binary. First, we move all numbers where the least significant bit is 0 to the left and all others to the right. This naturally separates the evens and odds. We then move to the next digit, where again 0 values stay put and values of 1 are pushed to the right. We keep doing this for all binary digits until we end up with a sorted list.

2.5

The data is sorted within the data structure without the use of external memory from an additional data structure. O(1) auxiliary data.

Sorting cards in your hand by swapping cards until they are in the correct order. Here we don’t need to use additional cards or people to hold cards, we can just have a single person swap cards until the deck is sorted.

Two elements that are equal won't change their order within the data, so the ordering of equal elements stays the same.

Queuing for tables at a restaurant - say you have an array of data representing the number of people in a party, and you (as the host/hostess) want to sort them by the number of people in the party. Because the first party of 2 in the array got there first, you want to seat them before any other party of 2. Therefore, you want that party of 2 to be the first 2 out of the other twos in the array.

Q3

3.1

Map Operation

We are operating on each element of a collection (the list of restaurants) independently to create a new collection (the list of restaurant:cuisine pairs) of the same size, which is the exact definition of a map operation.

3.2

Reduce

Producing a single answer from the collection via an associated operation. In this case, the collection is the dishes on the menu, and the associated operation is a max comparison.

3.3

Pack

We want to filter all the dishes such that only the dishes that have the vegetarian flag remain. Because this is a filter operation, Pack would be the best choice.

3.4

Prefix

A parallel prefix sum operation would give you the cumulative wait times of each party, as for each party it would give you the sum of its wait time, plus the wait times of all the parties before it in the list.

3.5

Pack

We want to filter all the desserts such that only the desserts under $10 remain. Because this is a filter operation, Pack would be the best choice.

Q4

4.1

Sum up even digits in input[i].

4.2

node.up = node.left.up + node.right.up

4.3

root.down = 0

4.4

node.left.down = node.right.up + node.down

4.5

node.right.down = node.down

4.6

output[i] = leaves[i].up + leaves[i].down

Q5

5.1

Race condition
Potential deadlock
Data race

There is a chance for potential deadlock between the newSubmission() and getNextSubmission() methods. Because both methods need numUngradedLock and taLock, a thread running newSubmission() could acquire numUngradedLock on line 9, then another thread running getNextSubmission() could acquire taLock on line 52, which would leave the newSubmission() thread waiting for taLock when it reaches line 19 and the getNextSubmission() thread waiting for numUngradedLock on line 53. A similar deadlock would also be possible on the same locks would be possible between the newSubmission() method and the finalGrading() method.

There is also a chance for a data race in howManyUngraded(). Because howManyUngraded() doesn't require the numUngradedLock, howManyUngraded() can read numUngraded while the value of the field is being changed, either on line 18 or line 62. Because two threads can potentially read and write numUngraded at the same time, this is a data race. Any data race is also considered as a race condition, and since there

is shown to be a data race, we can conclude that there is also evidence of a race condition.

5.2

LOCKS ADDED/RE-ORDERED AS A PART OF SOLUTION

```
public void newSubmission(int studentId) {
8      // Add a new submission and ensure each TA has something to grade
       taLock.acquire(); // Make sure each TA has something to grade.
9      numUngradedLock.acquire();
10     if (howManyUngraded() > submissions.length) {
11      numUngradedLock.release();
        taLock.release();
12      throw new UhhSystemIsBrokenException();
13     } else if (howManyUngraded() == submissions.length) {
14      System.out.println("Sorry, please try submitting again later :(");
15     } else {
16      submissionsLock.acquire(); // Add this submission
17      submissions[(nextSubmission + numUngraded) % submissions.length] = studentId;
18      numUngraded = howManyUngraded() + 1;
20       for (int i = 0; i < tas.length; i++) {
21        if (tas[i] < 0) {
22          getNextSubmission(taId);
23        }
24       }
25
26       submissionsLock.release();
27     }
28     numUngradedLock.release();
        taLock.release();
29   }

public int howManyUngraded() {
       numUngradedLock.acquire();
72     int temp = numUngraded;
       numUngradedLock.release();
       return temp;
73   }
```

Q6

6.1

First, make an undirected graph using the list of city pairs. For a pair (cityA, cityB), cityA and cityB would be vertices, and there would be an edge connecting cityA to cityB. Then, run a modified version of Kruskal's Algorithm on the undirected graph with all edge weights being equal to some arbitrary value, where the algorithm would stop running once the number of edges accepted is greater than the number of vertices - 1 OR once all the edges have been visited. The number of empires would be the final number of disjoint sets once Kruskal's algorithm has terminated.

6.2

First, make an undirected graph using the roads for Hansa as edges, and the cities as vertices. Then, run BFS V times (V being the number of cities), using each city as a starting point. For each starting vertex, record the longest path from that BFS traversal. After all the BFS traversals are complete, the vertex with the shortest longest path from their BFS will be the best for landing the spaceship in.

More efficient solution: Hansa is a tree, since it is a connected graph and does not have cycles. The question is really asking which node we need to pick from the graph so that the resulting tree has the minimum height.

Construct the graph and find all cities in Hansa with degree of one (aka leaves). For each leaf in the graph, we set a pointer for the leaf. Then we let the pointers "move up the tree" at the same speed (traversing up one node at a time). When two pointers meet, we combine the two pointers into a single pointer. We keep letting the pointers traverse up the tree until the last two pointers meet. The node that the final pointer points at should be the city that the spaceship should land on. There's also an edge case in which we are left with only two pointers at the end (two nodes) in the graph, and in that case, either node can be the solution.

Intuition behind this algorithm: any node that has a degree of 1 cannot be the answer (when we have more than two nodes in the graph). Proof by contradiction: if such node **v** is the root, suppose the longest path of the tree from **v** to a leaf node **f** is length **d.** In other words, **d** is the height of the tree. Consider picking **v'**, where **v'** is the only neighbor of **v** (since **v** only has a degree of 1), to be the root of the tree, in this case, **v'** to the leaf node **f** is length **d - 1,** and **v'** to **v** is distance 1, thus the height of the tree with root **v'** has a

minimum height of at most **Max(1, (d - 1)) = (d - 1)**, which is smaller than the height of the tree **d** if the root is **v**. Thus contradiction.

With the knowledge that any nodes with degree of 1 cannot be the answer, it's easier to solve the problem. Each iteration, we remove all the nodes with degree of 1, since they cannot be the answer. Once we've done that, the original graph **G** is transformed to the new graph **G'.** And we know the minimum height tree of graph **G** = minimum height tree of graph **G'** + 1. In other words, if we know the answer to **G'**, then we know the answer to **G**. Thus through induction, we can continue our logic on each iteration to remove all nodes with degree of 1, and we will derive the final answer (the ending condition will be when there are less than or equal to 2 nodes left in the graph).

6.3

Part a) There's no assumption that the graph constructed from Keshuth actually contains cycles or not. If it does, and if you can visit the same city more than once in a single path, then the solution is **undefined**, since there are infinitely many paths from Windijef to Richushi (you can go through a cycle infinite many times before reaching the destination.)

Otherwise, suppose the graph still contains cycles, but on a single path, you cannot visit the same city more than once. The solution is as follows:

Run a modified version of DFS with **recursive backtracking**. For each step of the exploration, keep track of a list of the nodes on the path from Windijef up to that node. At the same time, maintain a set of visited nodes so far. Once you are done with dfs for a particular node, remove the node from the list of the nodes of the path, and remove the node from the visited set as part of the backtracking process. If you've reached Richushi, append the list of nodes you've visited along the way to the answer, and then terminate. If you've no other nodes to explore in the next step, terminate directly.

Part b) Run a BFS on a graph representing the roads (edges) and cities (vertices) in Keshuth, with a starting vertex of Windijef. Once Richushi is found in the BFS, follow the

path fields of the vertices backwards to Windijef and reverse the answer. The result will be a path that takes the least number of days.

Part c) Run Dijkstra's Algorithm on a graph representing the roads (edges) and cities (vertices) in Keshuth, with a starting vertex of Windijef, and the edge weights being the time it takes to travel between two cities. Once the algorithm is complete, follow the path fields of the vertices backward to Windijef and reverse the answer. The result will be a path that takes the least amount of time to travel.

6.4

Create a directed graph representing the suggestions from the tourism bureau, with the vertices being museums and the edges being the order in which the museums need to be visited. So for (MuseumA, MuseumB), the graph would have an edge pointing from MuseumA to MuseumB. Then, perform a modified Topological Sort on the graph, where instead of finding and updating one vertex with a 0 in-degree at a time, you remove and process all the nodes with a 0 in-degree at the beginning of the day, and those nodes would be the museums you would visit that day. In other words, if you had a graph with edges (MuseumA, MuseumB) and (MuseumC, MuseumD), you would always process MuseumA and MuseumC before MuseumB or MuseumD, because A and C had a 0 in-degree at the start of the day. The resulting ordering would be in an order where all the museums you can visit in a day would be grouped together.

Q7

7.1

NP

P

We know that **Dijkstra’s algorithm** can be used to solve the shortest path problem in a directed weighted graph, and it is a Polynomial Time Algorithm, which makes this problem P, and P is inside of NP.

7.2

NP

P

We know that **DFS** can be used to solve the Cycle Finding problem in a graph, and it is a Polynomial Time Algorithm, which makes this problem P, and P is inside of NP.

7.3

None of these
This is the halting problem.

7.4

NP
P
Parallel prefix is Polynomial Time and the number of processors does not matter.

7.5

NP
NP-complete
This problem defines the **Hamiltonian Path Problem**, which we know is NP-complete.