Summer 2023 Final

CS 245: Summer 2023
Professor Prakash

Due: Thursday August 10 by 11:59pm PST

Instructions

Navigate to your local version of the Assignments repo. Download the starter code from the Assignments repo on Github by running the following command: **git pull origin main**. Make a copy of the finals folder and move it to your workspace. Also download this document. You will need it for the written portion of the exam.

Carefully read the instructions below to understand what files you need to modify for each question. This exam has a mix of coding and non-coding questions. For the coding portions of the exam, please modify the relevant files that are specified in each question. Any written non-coding answers should be included in **this document** in the appropriate section.

This is an individual assignment. You are not allowed to collaborate with any of your fellow classmates. We will not be answering any conceptual questions related to final either on Piazza, during lecture or office hours. We will only respond to clarifying questions or comments that point out potential bugs in the problems listed below.

You are allowed to reference any class slides, code examples, past assignments and lecture recordings to help you formulate your answers. You cannot use the internet or AI tools to crowd-source your solution in any way. **Any** form of cheating will get you a 0 and referral to the Dean's office with no exceptions.

Late Policy

The final is due on Thursday August 10 by 11:59 PM PST. **You cannot use any slip hours on this assignment**. For every 6 hours late you submit the assignment, we will reduce the maximum possible score you can earn by 25%. Any submissions turned in after August 11 will not be graded.

Grading

We provide all the test cases for the coding section of the exam. There are no hidden test cases. For each question, we have listed the corresponding test cases and their point value. We have also provided an autograder that you can use to verify your solutions (see the Autograder

section below). You will only know your score on the non-coding portion of the exam after we have looked at your individual answers.

Autograder

On Gradescope, we have setup two autograders that you can use to verify the correctness of your code. When you login into Gradescope, you will see two assignments: Final:Lists and Final:Trees.

The autograder under **Final: Lists** can be used to test the following questions:

1. Lists of Fun: Questions 1- 3
2. Question 8
3. Question 10
4. Extra Credit Question

To submit to the autograder, create a folder called **lists** that contains only Solutions.java from the lists package and SList.java. Upload a zipped version of the folder to run the autograder.

The autograder under **Final:Trees** can be used to test the following questions:

1. Questions 4-5

To submit to the autograder, create a folder called **trees** that contains only Solutions.java from the trees package. Upload a zipped version of the folder to run the autograder.

If your code passes the test cases locally, it will pass the autograder tests. That being said, it is your responsibility to make sure the autograder can process your submission. If your code has compile or runtime errors, the autograder will fail. If you upload the folder in the wrong format, the autograder will fail. We will be using the autograder to grade the coding portion of the final. If the autograder errors on your submissions, your score on the coding portion will be capped at **33%**.

Submission

Upload **both** your coding and written portions of the exam to your private repo on Github.

Lists of Fun (17 points)

For questions 1-3, you'll be modifying files in the lists package.

1. Python lists support slicing. For example, if x represents a list that contains the elements [0, 7, 5, 6, 15, 23, 9] and we call x[0:3:1], a new list that has [0, 7, 5] will be returned. In general, the syntax for slicing in Python is **x[start_idx : end_idx : step_size]**. The step size can also be a negative value. For example, if the step size was -1 and we called x[3:0:-1], the output would be [6, 5, 7].

   In other words, if we denote the step size as n, the slice operation includes the element at the starting location and then every nth element after that, up to **but not** including the last element.

   Java lists do not support slicing. So, we want to define a function that mimics this behavior. In **Solutions.java**, implement the function **slice** that takes in a List, the starting index, end index and step size and returns a sliced list. The length of the list is not guaranteed to be a multiple of the step size. If there are no elements to be returned, return an empty list. You can assume the step size is non-zero and that the start and end index are both non-negative. **(5 points)**

   Test Cases in SolutionsTest.java
   - **testSliceForward**: .5 point
   - **testSliceBackward**: .5 point
   - **testSliceForwardMultiple**: 1 point
   - **testSliceBackwardMultiple**: 1 point
   - **testSliceEmpty**: 1 point
   - **testSliceAll**: 1 point

2. When traversing an unsorted Linked List that stores numbers, we want to merge consecutive nodes that have the same value into one node, where the value of the merged node is the sum of nodes that were merged. For example, if the input is 1 -> 2 -> 2 -> 2 -> 3 -> 8, the output should be 1 -> 6 -> 3 ->8. There is no chained merging. For example, if the input is 1 -> 2 -> 2 -> 2 -> 6 -> 8, the output should be 1 -> 6 -> 6 -> 8, and not 1 -> 12 -> 8.

In the file **SList.java**, implement **mergeNodes** that supports the functionality described above. Your function must modify the list directly **(7 points)**

Test Cases in SListTest.java

- **testNoMerge**: 1 point
- **testSingleMerge**: 1 point
- **testHeadMerge**: 2 point
- **testMultipleMerge**: 3 point

3. We define the intersection of two Linked Lists to be another Linked List that contains only the values that are present in both lists. For example, if our two Linked Lists were 1 - > 5 -> 7 -> 9 and 1 -> 2 -> 3 -> 4 -> 8 ->9 -> 10, the output would be another Linked List that was 1 -> 9. The relative ordering of elements is important. In the example above, 9 -> 1 would not be a valid output.

   In **SList.java**, implement the function **linkedIntersection** that returns the intersection of two sorted Linked Lists. For simplicity, you can assume that there are no duplicate values in the input Linked Lists. Return an empty Linked List if there is no intersection between the two Linked Lists. The memory complexity of your algorithm must be O(1). The only extra memory you can allocate is for the Linked List that your function returns. You can add any helper functions or extra attributes to the SList class that you deem necessary to meet these constraints. **Memory complexities that are worse than O(1) will receive no credit.** (**5 points)**

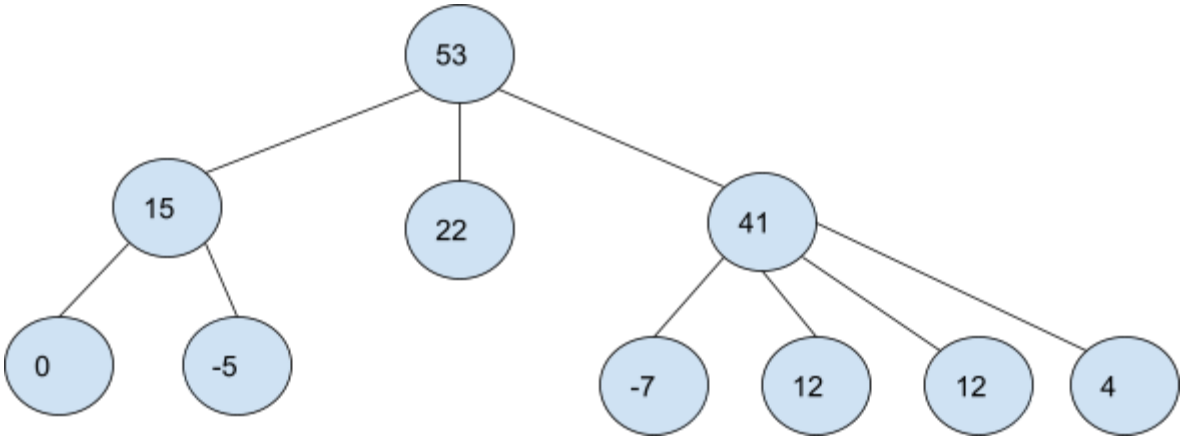
   Test Cases in SListTest.java

   - **testNoIntersection**: 1.5 point
   - **testAllIntersection**: 1.5 point
   - **testSomeIntersection**: 2 point

See the Forest (Amongst the Trees) (14 points)

For questions 4 and 5, you'll be modifying files in the trees package.

4. In **Solutions.java,** implement the function **reverseLevelOrder** that returns a list that stores values of the nodes that would be returned in reverse level order. This means that the nodes that are level k would have their values inserted into the output list before the nodes at level k - 1. For example, suppose our tree looked like:



The reverseLevelOrder method would return a list that contains the values [4, 12, 12, -7, -5, 0, 41, 22, 15, 53].

The tree that is passed into this function is not necessarily a Binary Tree **(7 points)**

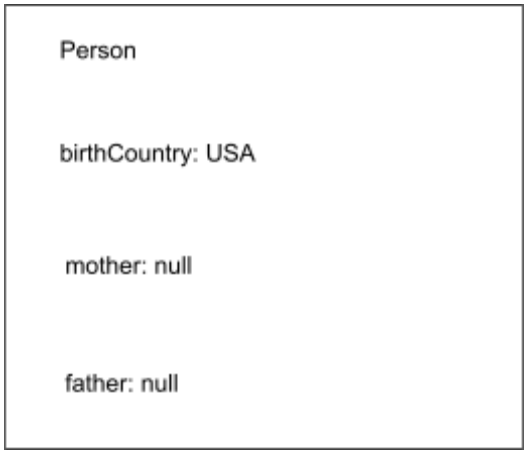
Test Cases

- **testBinaryTree**: 2 point
- **testLine**: 1 point
- **testVaryingDepths**: 4 point

5. Suppose we have the following Person class:

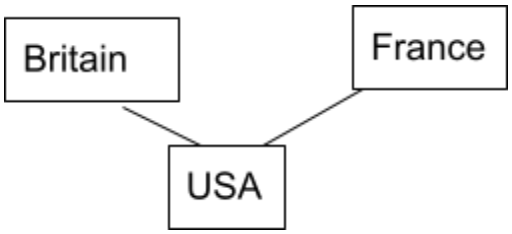
```
public class Person{
  String birthCountry;
  Person mother;
  Person father;
}
```

We'd like to know the person's heritage. For example, suppose we had a Person instance that looked like:



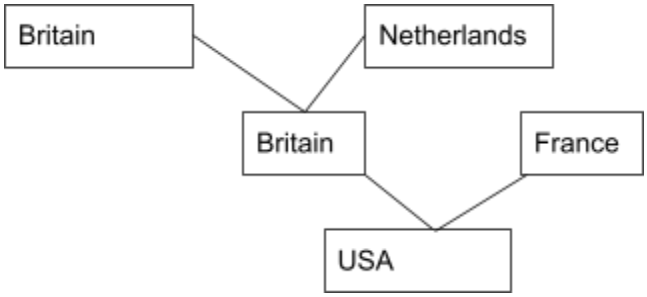
Since we don't know anything about the Person's ancestors, we'd say this person's heritage is 1.0 USA.

However, suppose we knew something about the Person's family tree and it looked like



In this case, we'd say the person has a heritage that is .5 Britain and .5 France.

Now suppose the family tree looked like:



The heritage of the person would now be .25 Britain, .25 Netherlands and .5 France. Notice that the person is not .5 Britain because the heritage of one its parents is only .5 Britain.

In **Solutions.java**, implement the function **nationalHeritage** that returns a HashMap, where the keys are Strings representing the countries, and the values are decimals representing the heritage tree. For simplicity, assume the father and mother attributes are either both null or both non-null. For example, we will never have a case where the mother attribute is null but the father attribute is not null. **(7 points)**

Credit: Question borrowed from UC Berkeley

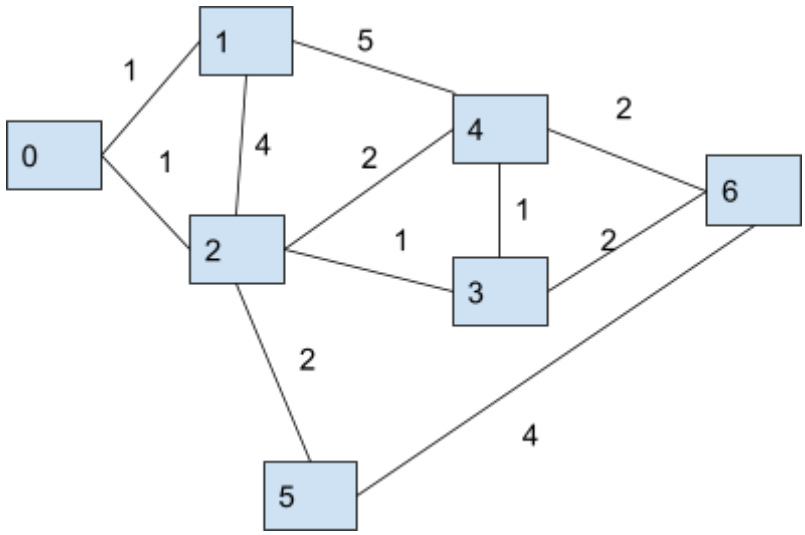
Test Cases

- **testBasicHeritage**: 1.5 point
- **testBasicHeritageV2**: 1.5 point
- **testBasicHeritageV3**: 4 point

The Graph Life (7 points)

This section has no coding questions.

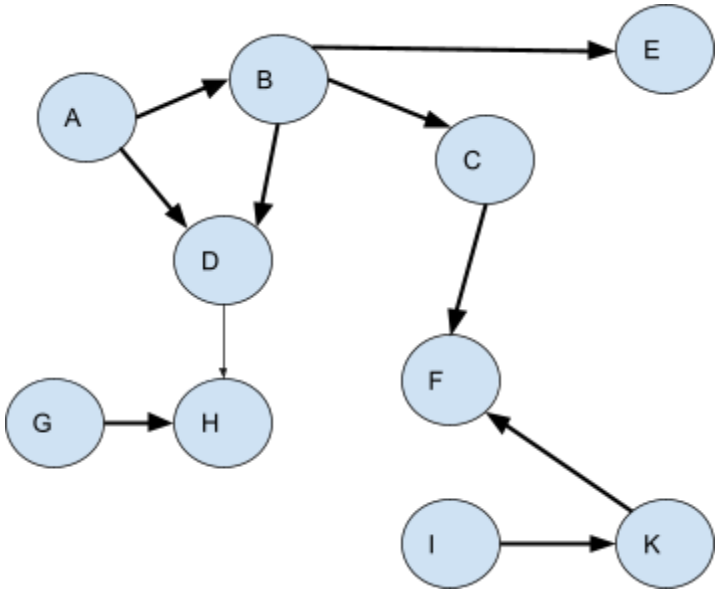
6. Suppose we have the undirected graph shown below. Provide you answers to 6a and 6b below.



a). Suppose the edges (0, 1) and (0, 2) have already been added to the MST. If we are running Prim's algorithm, which edge would get added next? If there are multiple possible answers, list all of them below. Be sure to justify your answer. **(2 points)**

b). Suppose the edges (0, 1) and (0, 2) have already been added to the MST. If we are running Kruskall's algorithm, which edge would get added next? If there are multiple possible answers, list all of them below. Be sure to justify your answer. **(2 points)**

7. Given the graph below, provide a valid topological sort of the nodes. If there is no possible topological sort, say so and explain why. If there are multiple possible topological sorts, you only need to provide one. **(3 points)**



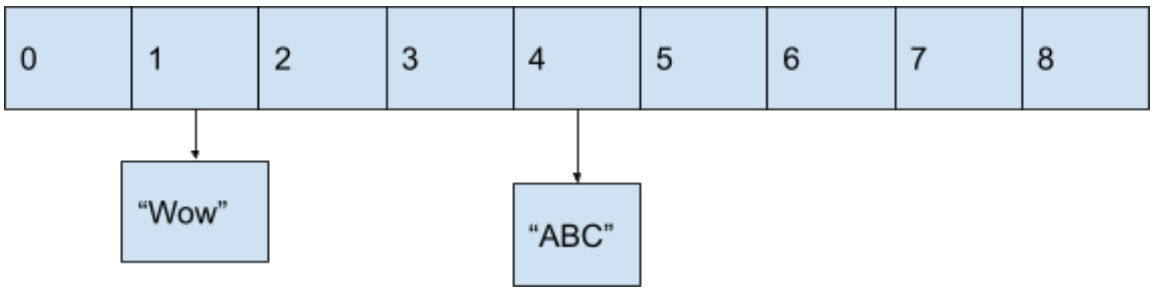
Don’t Make a Hash of Things (9 points)

For question 8, you will be modifying files in the lists package.

8. In **Solutions.java**, implement the function **groupAnagrams** that takes in a list of Strings and returns a List of sets, where each set contains the Strings that are anagrams of each other. For example, if the input is [“abcd”, “a”, “cabd”, “ab”, “ba”], the output should be [{“a”}, {“ab”, “ba”}, {“abcd”, “cabd”}]. You can assume there are no duplicate strings in the input. The ordering of the sets does not matter. **(5 points)**

   Test Cases in SolutionsTest.java
   - **testSingletons**: 1
   - **testOneGroup**: 1 point
   - **testMultiple**: 3 points

9. Suppose we have a HashTable that has the following state. We now want to insert the String “Hello World” into the HashTable. An anonymous classmate of yours wrote a hash function that determined the hash value of “Hello World” is 577.



   a. Describe what the HashTable would look like if the HashTable employed Separate Chaining. Explain your answer in 1-2 sentences. **(1 point)**

   b. Describe what the HashTable would look like if the HashTable employed the Open Addressing technique of Linear Probing. Explain your answer in 1-2 sentences. **(1 point)**

c. We saw we could use Binary Search Trees to do range queries. Can we perform range queries on HashTables? Why or why not? Explain your answer in 1-2 sentences. **(2 points)**

Get Your Sort in Order (13 points)

For question 10, you will be modifying files in the lists package.

10. In **Solutions.java**, implement the function **mergeAll** that accepts a list of sorted lists and returns one final sorted list after merging all of them into one. For example, if the sorted lists are [5, 9, 12, 14], [5, 5, 6, 6], [1,2,3,4] the output should be [1,2,3,4,5,5,5,6,6,9,12,14]. You cannot make any assumptions about the lengths of the lists or how many lists you need to sort. The input will contain at least one list.

    If there are k sorted lists to be merged, the memory complexity of your algorithm cannot be worse than O(k). Any solutions that are worse than O(k) in terms of memory complexity will receive no credit. **(9 points)**

    Test Cases in SolutionsTest.java
    - **testMergeEvenLists**: 1
    - **testMergeOddLists**: 2 point
    - **testMergeEvenLarge**: 3 points
    - **testMergeOddLarge**: 3 points

11. For each of the following scenarios, explain what sorting algorithm would be the most appropriate and why. You do not actually need to implement anything for this question. Be sure to justify your answers. Your explanation should mention the runtime complexity of the algorithm you chose for the following scenario.
    a. You want to sort students in ascending order in terms of height. **(2 points)**

b. Most of the elements are already sorted except for K randomly chosen elements, where K is way smaller than the length of the input. **(2 points)**

Data Structure Design (10 points)

12. Recall that when dealing with arrays, we could use indexing to directly retrieve an element. However, with Linked Lists we lost that capability. As a result, retrieving an element at index k in Linked Lists runs in O(N) time. Alec Smart is building a Singly Linked List and wants to develop an efficient way to retrieve, delete and insert nodes. To do so, Alec decides to augment the SList class with a HashMap called nodeMap, where the keys are integers to represent the node indices and the values are the actual SList nodes. He calls his new implementation of the SList class FastSList.

```
public void insert(int i, T value) {
   if (i < 0. || i > listSize) {
      throw IndexOutOfBoundsException("Out of bounds");
   } else if (i == 0) { // Add to front
      addToFront(value);
   }  else if (i == listSize) {
      addToBack(value); // add to back
   } else {
     // Add node with new value between two existing nodes
     ListNode newNode = new ListNode(value);
     ListNode nextNode = nodeMap.get(i + 1);
     ListNode prevNode = nodeMap.get(i - 1);
     addBetween(prevNode, nextNode, value);
   }
   nodeMap.put(i, newNode);
   listSize += 1;
}
```

Alec then implements the insert function as part of the FastSList class as shown above and claims that the method correctly supports insert operations runs in O(1) time. Is Alec's claim true? You can assume that **addToFront, addToBack and addBetween** update **only** the ListNode pointers correctly. You can also assume there are no compile or runtime errors with the code. **(5 points)**

13. In class, we talked about how you would implement Djikstra's Algorithm using a Priority Queue that was powered by a binary min-heap. Suppose instead of using a Priority

Queue, we decide to use a HashMap that maps each vertex to its priority (i.e distance from the source node). What would be the runtime of Djikstra's Algorithm in this scenario? Be sure to define any variables in your final formula. Hint: It would be helpful to fill in the table below **(5 points)**.

Operation	Times Performed	Single Execution Runtime	Total
findMin			
removeMin			
updatePriority			

Extra Credit (7 points)

For this problem, you will implement the function **bowlingPins** in Solutions.java that is part of the lists package.

Question: In a modified game of bowling, each pin is in a straight horizontal line. This means the bowler can see all the pins. Each pin has a numeric value associated with it. In this game of bowling, on each roll a player can either knock down exactly 1 pin, 2 pins or no pins. If the player hits 1 pin, their score gets incremented by the value of the pin. If the player hits two pins, the number of points earned is $V_i * V_{i+1}$ where V_i is the value of pin i. Once a pin gets knocked down, it cannot be used again.

We represent the values of the pins as an array of integers. For example, suppose the pin values were [4, 9, 2, 3, -1]. The optimal strategy would be to knock the 4 and 9 pin together (36 points), hit the 2 and 3 pin together (6 points) and avoid the -1 pin for a total of 42 points.

Assuming the bowler has an unlimited number of rolls, implement the **bowlingPins** function that returns the maximum score a player can earn given the setup of the game. Your function must run in O(N) time where the N is the number of pins. The memory complexity must be O(N) as well. Algorithms with worse runtime and memory complexities will receive no credit.

Test Cases in SolutionsTest.java

- **testECV1: 3 points**
- **testECV2: 1 point**
- **testECV3: 1 point**
- **testECV4: 2 points**

Congratulations. You are officially done with this class! :)