CSE 373 20su: Exam II Sample Solution

Note: This exam was originally offered via Gradescope.

1. Instructions

- This exam is written to be completed in 1-2 hours. To maximize flexibility, it will be available for 48 hours. **It is due strictly at 11:59pm PDT on Saturday, August 22.** You may not use late days on this exam.
- You may work in multiple sessions, and submit as many times as you'd like (only your last submission will be graded).
- You may work alone, or in groups of up to 8 (and submit a single exam). Group submission instructions:
  - First, have one person click "Save Answer" on any question. Then, scroll to the very bottom and click "Submit & View Submission".
  - You can now add your group members in the upper right corner, just like a programming assignment.
  - Then, you can click "Resubmit" to edit your answers as many times as you want, with the same group.
  - Be careful not to have two people edit answers at the same time (Gradescope may lose something). We recommend coordinating through a different platform and having one person submit for the whole group.
- This test is open-note and open-internet. However, you are not permitted to share these questions or get help from anybody not enrolled in 373 20su, such as former students.
- During the exam, you may ask clarification questions on Piazza, in office hours, or via email to cse373-staff@cs.washington.edu. Course staff will not answer questions about course concepts or give hints on specific exam questions.
- Sentence estimates are just to clarify our expectations. You will not be penalized for writing more or less.
- Each question is annotated with approximate corresponding learning objectives from lecture. This is purely extra information and only intended to make the link between what you've learned and what we're testing clearer: don't read too much into them or worry whether you've adequately demonstrated the skill! :)
- Any significant exam clarifications will be aggregated in this Piazza post.

2. Priority Queues & Heaps [8 pts]

Learning Objectives: Trace the removeMin(), and add() methods, including percolateDown() and percolateUp() (LEC12), Identify use cases where the PriorityQueue ADT is appropriate (LEC12), Describe how a heap can be stored using an array, and compare that implementation to one using linked nodes (LEC13)

For each of the following statements, mark whether it is ALWAYS, NEVER, or SOMETIMES true.

2.1. Heaps vs. AVL Trees [2 pts]

An AVL tree containing n elements has a smaller height than a binary min-heap containing n elements.

◯ ALWAYS

◯ NEVER

◯ SOMETIMES

Solution:

NEVER. A heap must always be a ”complete” tree (the Heap Structure Invariant), which is the most compact possible binary tree of n elements (with exactly $\log_2 n$ levels). An AVL tree with n elements could also be a ”complete” tree to have the same number of levels, but it couldn't possibly have *fewer* levels.

2.2. Heap Size [2 pts]

A binary min-heap with h levels contains $(2^h) - 1$ elements. Count the root node as the first level.

◯ ALWAYS

◯ NEVER

◯ SOMETIMES

Solution:

SOMETIMES. Due to the Heap Structure Invariant, a heap needs to be a ”complete” tree, meaning all but the last level must be full and the last level must fill from left to right. If the last level is completely full, a heap of h levels will have $(2^h) - 1$ elements, but otherwise

2.3. Min-Priority Queue ADT [2 pts]

Consider two data structures that implement the min-priority queue ADT storing integers (assume these are not extrinsic priority queues so there is no separate priority from the number itself, and note that the implementations of the data structures aren't specified): one where we insert 1, 2, 3, 4 (in order), and one where we insert 4, 3, 2, 1 (in order). If we then remove all the elements from both structures, the order in which we remove elements will be different.

◯ ALWAYS

◯ NEVER

◯ SOMETIMES

Solution:

NEVER. Remember that while a data structure defines the implementation, it is the ADT (in this case, the Min-PriorityQueue ADT) that defines the ”contract” of behaviors a client can expect. For Min-PriorityQueues, the removeMin method must always remove the minimum element, regardless of implementation or how the elements were inserted.

2.4. Underlying Implementation [2 pts]

In class, we studied how to implement a binary min-heap by representing it in an underlying array. An implementation with the same heap representation but stored a doubly-linked list (with a pointer to the back) would have the same asymptotic runtime as the array implementation for any call to removeMin().

◯ ALWAYS

◯ NEVER

◯ SOMETIMES

Solution:

SOMETIMES (we also awarded full credit for NEVER). Since the array representation of a heap requires accessing arbitrary elements in the middle of the array, representing the same data in a linked list could have the same runtime for the best case (no percolation necessary) but would have worse runtime when percolations are involved. We gave full credit for NEVER, under the assumption that it's never possible to store a heap in a linked list in such a way that *any* call would always be the same asymptotic runtime.

3. Heap Invariants [12 pts]

Learning Objectives: Describe the invariants that make up a heap and explain their relationship to the runtime of certain heap operations (LEC12), Trace the removeMin(), and add() methods, including percolateDown() and percolateUp() (LEC12)

Suppose we modify the Heap Structure Invariant such that the bottom level is not required to be filled from left to right. The following is a high-level description of how the `add` and `removeMin` methods would work in the new heap:

- `add(item)`: find the first null entry on the bottom level (or create a new row if the bottom row is full), insert item there, then percolate up
- `removeMin()`: replace the root node with a "null value", then repeatedly swap that "null value" with its smallest child until it has no children.

In this problem, assume the heap is implemented using an underlying array. When doing runtime analysis, assume the array never needs to resize.

3.1. Heap Invariant [2 pts]

Does this new heap still meet the Heap Invariant that requires nodes to have priorities less than or equal to their children's?

◯ Yes

◯ No

Solution:

Yes. Given the operations listed, it is not possible to have a node with a priority greater than its children's, because it would be swapped in either operation that could place it there.

3.2. Heap Structure Invariant [4 pts]

Does this new heap meet the other half of the Heap Structure Invariant that requires all but the bottom level to be full?

◯ Yes

◯ No

Solution:

No. Since we only swap a "null value" with its smallest child until it *has no children*, and not necessarily when it's on *the last level*, it's possible to create a heap that is not a complete tree. For example, suppose we started with a heap as a perfect tree. Given the right ordering of values within, the first two removeMin operations could remove the two rightmost nodes on the last level as normal. However, then it would be possible for the next removeMin operation to swap a null value until it is in the parent's position of those two removed nodes – creating a hole in the second-to-last level before the last level has been fully depleted.

3.3. add Best Case [1 pt]

Give a simplified Big-Theta for the runtime of the new add method in the best case. Write the class only: for example "$n \log n$" instead of "$\Theta(n \log n)$".

Solution:

1. In the best case, the add method would simply insert a new element and do no swapping or percolating, as normal.

3.4. add Worst Case [2 pts]

Give a simplified Big-Theta for the runtime of the new add method in the worst case. Write the class only: for example "$n \log n$" instead of "$\Theta(n \log n)$".

Solution:

n. Given that the new heap does not need to maintain a complete tree, it is possible for all n nodes in a heap to form a degenerate tree (effectively a linked list), which the add operation might have to percolate all the way up.

We awarded full credit for $\log n$ for groups that answered "Yes" in 3.2.

3.5. removeMin Best Case [1 pt]

Give a simplified Big-Theta for the runtime of the new removeMin method in the best case. Write the class only: for example "$n \log n$" instead of "$\Theta(n \log n)$".

Solution:

1. In the best case, the removeMin method would simply remove the last element and do no percolating or swapping, as normal.

3.6. removeMin Worst Case [2 pts]

Give a simplified Big-Theta for the runtime of the new removeMin method in the worst case. Write the class only: for example "$n \log n$" instead of "$\Theta(n \log n)$".

Solution:

n. Given that the new heap does not need to maintain a complete tree, it is possible for all n nodes in a heap to form a degenerate tree (effectively a linked list), which the removeMin operation might have to percolate all the way down.

We awarded full credit for $\log n$ for groups that answered "Yes" in 3.2.

4. Pizza Party [10 pts]

Learning Objectives: [some objectives omitted to avoid giving away the solution], Identify whether algorithms are considered reductions (LEC22)

You're starting up a new pizza restaurant, and you need to decide what pizzas to put on your menu. Fortunately, with your wisdom as a CSE 373 graduate, you've come up with a model that will allow you to determine the optimal set of pizzas. Unfortunately, due to budget constraints, you will only be able to select 3 pizzas, and **no topping can be used on more than one pizza, but every pizza must have at least one topping**.

According to your model, every pair of toppings has a particular “pairing value”. For a pizza with T toppings, its “flavor value” is equal to the maximum sum of $T-1$ “pairing values” that connect all of its ingredients.

The following is **one example** set of pairing values for four ingredients:

	Pepperoni	Pineapple	Ham	Mushroom
Pepperoni	-	-	-	-
Pineapple	25	-	-	-
Ham	20	50	-	-
Mushroom	15	5	12	-

With these pairing values…

- A pizza with pepperoni and pineapple has flavor value 25.
- A pizza with pepperoni, pineapple, and ham yields a flavor value of $25 + 50 = 75$.
  - Note that this is not $25+50+20 = 95$, because only $T-1$ toppings can be considered for the flavor value of a pizza with T toppings.
- Using all 4 toppings on the same pizza yields $25 + 50 + 15 = 90$.
  - Note that this is not $25 + 50 + 20 = 95$ since 20 would not connect the mushroom topping.

4.1. Approach [8 pts]

Describe an approach to determine which toppings to use in each of the 3 pizzas on your menu that **maximizes the sum of their flavor values.** This will involve modeling the situation using the appropriate structure(s) and one of the algorithms we covered in class, potentially with modifications. Your approach should produce 3 sets of toppings (e.g. as a list of sets), and note that an optimal menu will end up using all of the toppings at your disposal. **Your approach should work for any set of toppings and their pairing values**, not just the example above. Assume the number of pizzas is fixed at 3, but there may be any number of toppings. (~3-6 sentences)

For full credit, your description must include:

(a) How you would construct the inputs to your algorithm from the situation.

(b) Which algorithm from class you are using.

(c) Whether you would need to modify that algorithm (and if so, how?)

(d) How to get the output sets from the algorithm’s output.

Solution:

There are many correct responses to this question. One sample solution:

Model the situation as a weighted, undirected graph. Create a vertex in the graph for each vertex, and create an edge between every pair of toppings weighted with their ”pairing value”. Then, run a modified version of Kruskal’s algorithm on this graph (including the initialization of a Disjoint Sets structure) with two major differences:

1. At every iteration of the algorithm, add the largest edge instead of the smallest (we are computing a ”maximal spanning tree”). 2. Stop the algorithm when the number of edges reaches V-3, instead of the normal V-1.

This algorithm will divide the graph into 3 separate subgraphs such that the sum of the ”maximal spanning trees” over them will be maximized (for intuition, you can think about how to optimally split the maximal spanning tree of the whole graph into 3 maximal spanning trees: removing the 2 smallest-weight edges will always maximize the sum of the resulting trees, which is exactly what this algorithm does by stopping early). After running the modified algorithm, iterate through all vertices in the graph and insert them into 3 sets based on the result of calling `find` with them on the DisjointSets structure. Return those 3 sets.

4.2. Reductions [2 pts]

Is your solution in part 4.1 a reduction?

◯ Yes

◯ No

Solution:

The answer to this question varied depending on your solution in 4.1. The example solution above can be thought of as a modification of Kruskal’s algorithm, which would not be a reduction.

5. Graph Traversals: Code [10 pts]

Learning Objectives: Implement iterative BFS and DFS (LEC15), Describe the Shortest Paths Problem, write code to solve it, and explain how we could use a shortest path tree to come up with the result (LEC15), Synthesize code to solve problems on a graph based on DFS, BFS, and Dijkstra's traversals (LEC16)

Suppose we have the following Java interfaces:

```java
interface Vertex {
}
interface Edge {
    public Vertex to();
    public Vertex from();
}
interface Graph {
    public List<Edge> edgesFrom(Vertex v);
}
```

And the following implementation of BFS, as described in lecture. For the remainder of this question, assume that the start vertex is guaranteed to be contained in the graph.

```java
void bfs(Graph graph, Vertex start) {
    Queue<Vertex> perimeter = new Queue<>();
    Set<Vertex> visited = new Set<>();
    perimeter.add(start);
    visited.add(start);
    while (!perimeter.isEmpty()) {
        Vertex from = perimeter.remove();
        for (Edge edge : graph.edgesFrom(from)) {
            Vertex to = edge.to();
            if (!visited.contains(to)) {
                perimeter.add(to);
                visited.add(to);
            }
        }
    }
}
```

For both parts of this question, you will write compiling Java code (not pseudocode) to perform the specified task. You are allowed to use additional data structures from Java's standard library (don't worry about import statements).

5.1. Maximum Level [4 pts]

Let the "level" of a node be the minimum number of edges between it and the start. Given a graph and a start vertex, we want to compute the maximum level among the other vertices in the graph. For example, if we had a graph with one vertex at level 1 from the start, two vertices at level 2, and two vertices at level 3, the correct result would be 3 (it doesn't matter that there are multiple vertices tied for max level because we only care about the level itself).

Copy the above BFS code and modify it to return the maximum level. We recommend typing your solution in an IDE and pasting it here afterward.

Solution:

There are many correct solutions. One sample solution:

```
int bfs(Graph graph, Vertex start) {
    Queue<Vertex> perimeter = new LinkedList<>();
    Set<Vertex> visited = new HashSet<>();

    Map<Vertex, Integer> distTo = new HashMap<>();

    perimeter.add(start);
    visited.add(start);

    distTo.put(start, 0);
    int maxLevel = 0;

    while (!perimeter.isEmpty()) {
        Vertex from = perimeter.remove();
        for (Edge edge : graph.edgesFrom(from)) {
            Vertex to = edge.to();
            if (!visited.contains(to)) {
                perimeter.add(to);
                visited.add(to);

                distTo.put(to, distTo.get(from) + 1);
                maxLevel = distTo.get(from) + 1;
            }
        }
    }
    return maxLevel;
}
```

5.2. Influencers [6 pts]

Now, suppose we want to use a graph to represent a social network. In our model, we will represent people as vertices and represent that two people are "friends" with an undirected edge between them. We will also make the assumption that every person is either an "influencer" or not, so in this question we'll modify the Vertex interface to add a boolean method that returns this property:

```
interface Vertex {
    public boolean isInfluencer();
}
```

We want to model the spread of a user's post in this social network to calculate how many "impressions" the post would get, where an impression is a single user seeing a single post. Assume the rules for posts spreading is as follows: normally, a post made by user A will only be seen by A's friends. However, any user in the graph marked as an "influencer" will also share the post if they see it. When an influencer shares the post, it is seen by all of *their* friends, and **we count it as a separate impression if the same person sees a post twice from two different friends.** Once an influencer has shared a post, they will not share it again. If user A sees the post shared from someone else, count it as an additional impression just like for any other user. Do not count user A as an impression for user A's original post.

Copy the above BFS code and modify it to return the number of impressions of a post made by the user whose vertex is passed in as `start`. We recommend typing your solution in an IDE and pasting it here afterward.

Solution:

There are many correct solutions. One sample solution:

```java
int bfs(Graph graph, Vertex start) {
    Queue<Vertex> perimeter = new LinkedList<>();
    Set<Vertex> visited = new HashSet<>();

    perimeter.add(start);
    visited.add(start);

    int impressions = 0;

    while (!perimeter.isEmpty()) {
        Vertex from = perimeter.remove();
        for (Edge edge : graph.edgesFrom(from)) {
            Vertex to = edge.to();

            impressions++;

            if (!visited.contains(to) && to.isInfluencer()) {
                perimeter.add(to);
                visited.add(to);
            }
        }
    }
    return impressions;
}
```

6. Graph Traversals: Analysis

Learning Objectives: Implement iterative BFS and DFS, and synthesize solutions to graph problems by modifying those algorithms (LEC15), Evaluate inputs to (and modifications to) Dijkstra's algorithm for correct behavior and efficiency based on the algorithm's properties (LEC16)

Consider the following implementation of DFS, called newDFS. The only modification we've made is placing a vertex in the visited set when we pop it from the perimeter, rather than when we add it to the perimeter:

```
void newDFS(Graph graph, Vertex start) {
    Stack<Vertex> perimeter = new Stack<>();
    Set<Vertex> visited = new Set<>();
    perimeter.add(start);
    visited.add(start);
    while (!perimeter.isEmpty()) {
        Vertex from = perimeter.pop();
        visited.add(from); // this is the line we moved
        for (Edge edge : graph.edgesFrom(from)) {
            Vertex to = edge.to();
            if (!visited.contains(to)) {
                perimeter.add(to);
            }
        }
    }
}
```

6.1. Visited Set [2 pts]

On the same graph, how would the size of the visited set compare between newDFS and DFS after finishing each algorithm?

- ◯ The size of the visited set would increase
- ◯ The size of the visited set would decrease
- ◯ The size of the visited set would stay the same

Briefly justify your answer: (∼1-2 sentences)

Solution:

The size of the visited set would stay the same. All the same nodes would be visited in the new algorithm, and because `visited` is a set, it cannot store duplicates.

6.2. Number of Iterations [2 pts]

On the same graph, how would the size of iterations of the while loop compare between running `newDFS` and running `DFS`?

- ◯ The number of iterations of the algorithm could increase or stay the same
- ◯ The number of iterations of the algorithm could decrease or stay the same
- ◯ The number of iterations of the algorithm would always stay the same

Briefly justify your answer: (∼1-2 sentences)

Solution:

The number of iterations of the algorithm could increase or stay the same. In the new algorithm, a node is not marked visited until it has been popped off `perimeter`, unlike the original where a node is marked visited when it is pushed onto `perimeter`. Therefore, if multiple nodes in a graph had edges to a particular node, that node could be pushed into `perimeter` multiple times before the first one got popped and it was marked visited to prevent further pushes. By placing multiple copies of a single node on the stack, the algorithm could run for more iterations.

6.3. Finding a Vertex [2 pts]

Suppose there is a particular vertex we are searching for that is successfully found by DFS (where found means it is popped from the perimeter). On the same graph, would newDFS be guaranteed to find the same vertex?

◯ Yes

◯ No

Briefly justify your answer: (∼1-2 sentences)

Solution:

Yes. Since the new algorithm would search all the same nodes as the original, it would be able to find any node found by the original.

6.4. Dijkstra's Algorithm [3 pts]

(This question is unrelated to newDFS)

Which of the following properties does Dijkstra's algorithm depend on for correctness? In other words, if the property did not hold for a run of Dijkstra's, which property could cause the algorithm to return an incorrect result?

◯ Dijkstra's algorithm never considers whether an edge could be a shorter path to a vertex that is already "known"

◯ Dijkstra's always visits the vertex with the shortest tentative path next

◯ Dijkstra's algorithm visits all vertices in the graph

Solution:

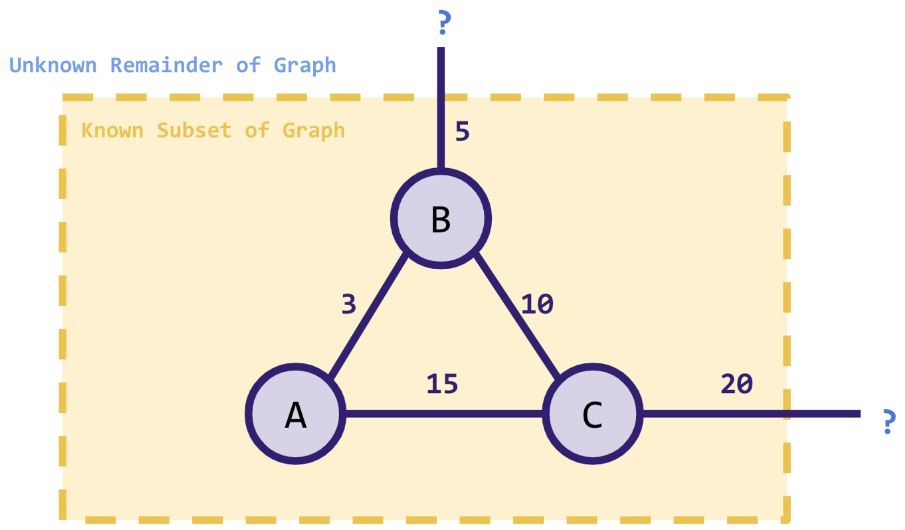
B: Dijkstra's always visits the vertex with the shortest tentative path next. Note that A does not need to be true for Dijkstra's to be correct, and in fact is not true for the pseudocode given in lecture. Because Dijkstra's will always consider the shorest tentative path next, it's okay for the algorithm to *consider* whether an edge to an already-known vertex could be a shorter path – because it will never end up being the case (our pseudocode in lecture does this, and simply relies on the fact that it will never perform an update in that case). C also does not have to be true for Dijkstra's to be correct, and isn't true on all graphs: if a graph is disconnected, Dijkstra's will not visit all nodes, but that can still yield the correct result (indicating that there is no possible path from the start). B is correct because Dijkstra's *must* go in order of shortest path next to yield the right answer – any other ordering could cause it to "commit" to a shortest path before the true shortest path to that node had been found.

7. Minimum Spanning Trees [10 pts]

Learning Objectives: Identify a Minimum Spanning Tree (LEC17), Explain why the Cut and Cycle properties must be true from the definition of an MST (LEC17)

Consider the following diagram, which shows a partial snippet (a subset of the vertices and edges) of a weighted undirected graph. More precisely, assume that there may be any number of additional edges and vertices in the rest of the graph that are not pictured, and that every edge connecting to a "?" symbol is connected to some unpictured vertex—but we don't know anything about the rest of the graph, so it's possible that both "?" placeholders are the same vertex. Assume the two edges with "?" symbols are the only edges crossing between the two portions. Do not assume anything else about the rest of the graph.

For each edge in the subset, indicate whether it will NEVER, ALWAYS, or SOMETIMES be in a minimum spanning tree of the graph.



7.1. Edge 3 (A – B) [2 pts]

- ◯ ALWAYS
- ◯ NEVER
- ◯ SOMETIMES

Solution:

ALWAYS. Consider a cut with B on one side and all other vertices on the other: by the Cut Property, edge 3 must be in the MST since it is the smallest crossing edge.

7.2. Edge 5 (B – ?) [2 pts]

- ◯ ALWAYS
- ◯ NEVER
- ◯ SOMETIMES

Solution:

ALWAYS. Consider a cut with A, B, and C on one side and all other vertices on the other: by the Cut Property, edge 5 must be in the MST since it is the smallest crossing edge.

7.3. Edge 10 (B – C) [2 pts]

- ◯ ALWAYS
- ◯ NEVER
- ◯ SOMETIMES

Solution:

ALWAYS. Consider a cut with C on one side and all other vertices on the other: by the Cut Property, edge 10 must be in the MST since it is the smallest crossing edge.

7.4. Edge 15 (A – C) [2 pts]

- ◯ ALWAYS
- ◯ NEVER
- ◯ SOMETIMES

Solution:

NEVER. Consider the cycle A–B, B–C, C–A. By the Cycle Property, edge 15 can never be included in the MST since it is the largest crossing edge.

7.5. Edge 20 (C – ?) [2 pts]

- ◯ ALWAYS
- ◯ NEVER
- ◯ SOMETIMES

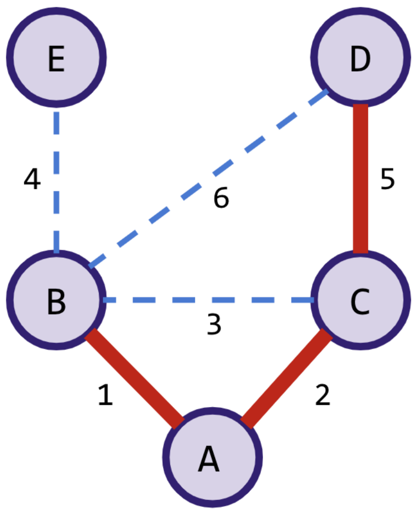
Solution:

SOMETIMES. For this edge, we must consider the possible structure of the rest of the graph. If edge 20 is the only edge that connects a single, otherwise disconnected vertex on the other side to the rest of the graph, then edge 20 must be included. However, suppose edge 20 and edge 5 connect to the same vertex in the unknown remainder of the graph – then, only edge 5 would be needed to connect the entire known subset and edge 20 would not be part of the MST.

8. Prim's & Kruskal's [10 pts]

Learning Objectives: Implement Prim's Algorithm and explain how it differs from Dijkstra's (LEC17), Describe Kruskal's Algorithm at a high level and explain why it works (LEC18)

Consider the following diagram, which shows an intermediate step of building up an MST. Thick, solid, red lines indicate edges that have been selected for the MST so far, and thin, dashed, blue lines indicate edges that have not.



8.1. MST Algorithms [4 pts]

Which of the following algorithms could this be an intermediate step of? Select all that apply.

☐ Prim's starting from A

☐ Prim's starting from C

☐ Prim's starting from D

☐ Kruskal's

☐ None of the above

Solution:

Only Prim's starting from D could produce this intermediate step. If Prim's were to start from A or C, in the previous step it would consider edge 4 before considering edge 5 because both would be connected to the partial MST being built up but edge 4 is smaller. If Kruskal's were running, it would consider edge 4 before edge 5 purely because it is a smaller edge.

8.2. Prim's Algorithm

Suppose you wanted to explain to your friend why it is correct for Prim's algorithm to consider only the weight of each edge in its update function, instead of also adding the distance to the edge's origin like in the update function of Dijkstra's algorithm. For each of the following descriptions of graphs, determine whether the graph could be used as an example to show why considering the full distance from the origin in Prim's algorithm is incorrect.

8.3. Prim's Algorithm: Graph Description 1 [2 pts]

A weighted undirected graph in which one of its valid SPTs (Shortest Paths Trees) is not also an MST (Minimum Spanning Tree).

◯ This graph could definitely be used as an example to show why considering the full distance from the origin in Prim's algorithm is incorrect

◯ This graph couldn't necessarily be used

Solution:

This graph could definitely be used. Note that Prim's algorithm with the full distance of each path instead of considering only edge weights is exactly the same as Dijkstra's algorithm, and Dijkstra's algorithm computes an SPT while Prim's algorithm computes an MST. Therefore, a graph where an SPT was not an MST could be used to show *why* Prim's considers only edge weights: if it *were* to take edge weights into account, it wouldn't end up with an MST.

8.4. Prim's Algorithm: Graph Description 2 [2 pts]

A weighted undirected graph with an MST (Minimum Spanning Tree) consisting of exactly the $V-1$ smallest edges in the graph.

◯ This graph could definitely be used as an example to show why considering the full distance from the origin in Prim's algorithm is incorrect

◯ This graph couldn't necessarily be used

Solution:

This graph couldn't necessarily be used. This description doesn't give enough information to know whether all SPTs in the graph are also MSTs, or if some of them are not. Therefore, we cannot be certain whether this graph could be used as an example to show why Prim's considers only edge weights.

8.5. Prim's Algorithm: Graph Description 3 [2 pts]

A weighted undirected graph with exactly one MST (Minimum Spanning Tree) and exactly one SPT (Shortest Paths Tree).

◯ This graph could definitely be used as an example to show why considering the full distance from the origin in Prim's algorithm is incorrect

◯ This graph couldn't necessarily be used

Solution:

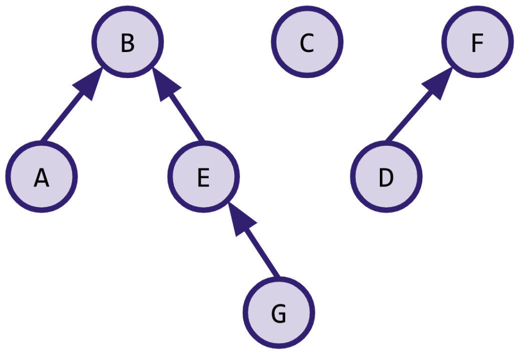
This graph couldn't necessarily be used. This description doesn't give enough information to know whether the single SPT in the graph is also the MST, or if they are different. Therefore, we cannot be certain whether this graph could be used as an example to show why Prim's considers only edge weights.

9. Disjoint Sets [5 pts]

Learning Objectives: Implement WeightedQuickUnion and describe why making the change protects against the worst case find runtime (LEC19), Implement path compression (LEC19), Implement WeightedQuickUnion using arrays and describe the benefits of doing so (LEC19)

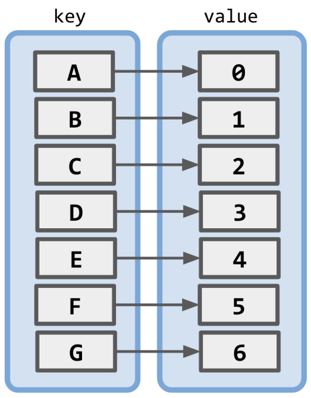
Consider the following WeightedQuickUnion data structure implementing the Disjoint Sets ADT as described in lecture, as well as the corresponding array implementation. Note that the data structure also maintains a map from each element to that element's index in the array.

Abstract Representation:



Implementation:

map



array

0	1	2	3	4	5	6
1	-4	-1	5	1	-2	4

9.1. Union [2 pts]

Write out the contents of the array after calling `union(G, D)`.

Solution:

[1, -6, -1, 5, 1, 1, 4] (if path compression is not being used)

or

[1, -6, -1, 5, 1, 1, 1] (if path compression is being used).

We accepted both answers for full credit since the question is somewhat ambiguous about whether to use path compression.

9.2. Path Compression [3 pts]

Suppose we implement path compression as covered in lecture. Write out the contents of the array after calling `find(G)` on the **original** array (not your solution in 9.1).

Solution:

[1, -4, -1, 5, 1, -2, 1]

10. Sorting [25 pts]

Learning Objectives: Define an ordering relation and stable sort and determine whether a given sorting algorithm is stable (LEC20), Implement Selection Sort and Insertion Sort and compare runtimes and best/worst cases of the two algorithms (LEC20), Implement Heap Sort, describe its runtime, and implement the inplace variant (LEC21), Implement Merge Sort, and derive its runtimes (LEC21), Implement Quick Sort, derive its runtimes, and implement the in-place variant (LEC22), Define a topological sort and determine whether a given problem could be solved with a topological sort (LEC22)

10.1. Multiple Sorting Algorithms [3 pts]

Java's built-in default sort implementation checks the type of the items being sorted and decides which sorting algorithm to use accordingly. If passed an array of 1000 integers, Java would use a variation of Quick Sort, but if passed an array of 1000 Point objects, Java would use Merge Sort.

Explain why it might be beneficial to use these two different sorting algorithms by identifying a property (or properties) that differ between the algorithms and why each would be most appropriate for the case described. (∼1-3 sentences)

Solution:

Merge Sort is stable, so it's preferred in cases where stability could matter to the user, like when sorting objects (even though the user may not care about stability, as a design decision this makes sense to form a useful default without the user having to specify). Quick Sort is generally faster in practice, so it's preferred in cases where stability can't possibly matter, like sorting primitive data types.

10.2. Topological Sort [2 pts]

Is it possible to find a topological ordering for a graph with unlabeled vertices?

◯ Yes

◯ No

Solution:

Yes. Topological sort only orders a graph based on the edges between vertices, and does not compare the values of vertices themselves – so no labels are required to still put vertices in a topological ordering.

10.3. Sorting Steps [5 pts]

The following shows a series of steps in order at certain points in a sorting algorithm, although there may be additional steps omitted in between. Which sorting algorithm(s) could be being run here? Select all that apply.

Initial Array:
`10, 5, 8, 3, 4, 2, 9, 1, 6, 7`
Some Later Step:
`5, 6, 7, 10, 9, 8, 4, 3, 2, 1`
Final Step:
`1, 2, 3, 4, 5, 6, 7, 8, 9, 10`

- ☐ Insertion Sort
- ☐ Selection Sort
- ☐ In-place Heap Sort
- ☐ Merge Sort
- ☐ In-place Quick Sort

Solution:

These steps can only come from running In-place Heap Sort – note that in the intermediate step, the sorted part of the array is being built up backward (the rightmost 4 elements), while the other part of the array is organized into a heap. At the end of the algorithm, the entire result is flipped to come up with the correct sorted array.

Because the final step shows the elements in ascending order, we know that is the ordering relation being used here. This cannot be insertion sort, because it would be impossible for the 7 and 6 to be inserted into the sorted part of the array before the 8 and 3. This cannot be selection sort, because it would be impossible for 1 not to be in the starting position after the first step. This cannot be merge sort, because it would be impossible for the 3 to make it to the right half – it would be contained in left-half subarrays for the entire algorithm, and only in the final step could it ever possibly be placed on the right half of the overall array. Finally, this also cannot be quick sort, because quick sort always proceeds by scanning through the elements to be partitioned and placing them into two halves using the original relative order they had. As a result, no matter what pivot selection strategy or how many steps of quick sort had occurred, it would be impossible for the 3 and 9 to swap their relative order in an intermediate step of the algorithm.

10.4. Quick Sort Worst Case [3 pts]

Suppose we implement Quick Sort using a pivot selection strategy where we choose the median of the first, last, and middle elements of an array (if the array is an **even** length, choose the element right before the middle). Give a 7-element array containing the values 1 through 7 (each appearing exactly once) that would result in worst-case behavior for Quick Sort with this pivot selection strategy.

Solution:

There are many possible solutions. One possible sample solution:

[1, 5, 6, 2, 4, 7, 3]

10.5. Comparing and Comparing Again [3 pts]

Which of the following sorting algorithms never compare the same pair of elements more than once? Select all that apply.

☐ Insertion Sort

☐ Selection Sort

☐ In-place Heap Sort

☐ Merge Sort

☐ In-place Quick Sort

☐ None of the above

Solution:

Insertion Sort, Merge Sort, and In-Place Quick Sort. We gave full credit for choosing or omitting Selection Sort, because it is ambiguous whether comparing elements to each other in the process of finding the minimum should count as a comparison. In insertion sort, only a single element is being considered for insertion at a time. In merge sort's combine step, elements are only compared to elements in the other subarray, and once those subarrays are merged, an element will never be compared to anything in the same array again. In quick sort's divide step, only a single pivot is being used for comparison at a time, and after being compared against all elements it is partitioned apart from the rest of the graph and not considered again. Finally, in in-place heap sort, a pair of elements may be compared against each other many times if the same element percolates in the heap multiple times.

10.6. Debugging Merge Sort [9 pts]

Consider the following **BUGGY** code intended to implement the merge portion of the Merge Sort combine step on arrays of some Element object. All we know about Element is that it implements the Comparable interface, but we don't know what data it stores or how compareTo is implemented. As described in class, it accepts two arrays (each assumed to be sorted, and assumed to have come from the left and right portions of the split array respectively) and should return a new array containing the elements in sorted order. Here, assume the overall Merge Sort algorithm should work for any size input array, be stable, and should run in $\Theta(n \log n)$ time. Assume that this method would be called many times by some other code that implements the rest of the Merge Sort combine step. You may assume the `left` and `right` arrays contain no null elements.

```
public Element[] merge(Element[] left, Element[] right) {
  Element[] result = new Element[left.length + right.length];
  int indexLeft = 0;
  int indexRight = 0;
  int indexResult = 0;
  while (indexLeft < left.length and indexRight < right.length) {
    if (left[indexLeft].compareTo(right[indexRight]) < 0) {
      result[indexResult] = left[indexLeft];
      indexResult++;
      indexLeft++;
    } else {
      result[indexResult] = right[indexRight];
      indexResult++;
      indexRight++;
    }
  }
  return result;
}
```

The above code contains at least one error and as many as three errors. For each error in the code, give: (1) its location in the code (please include a line number to help us find it if applicable); (2) a description of what can go wrong; and (3) a description of how you would change the code to fix the bug. Use a separate answer box for each error (if there are fewer than three errors in the code, leave the rest of the boxes empty).

Solution:

There are 2 bugs in the code:

(a) The while loop condition on line 6 finishes when either subarray is completely exhausted, but not when both are exhausted – so it is possible for all elements to be taken from left without any elements being taken from right. There are many possible ways to fix this. One possible solution would be adding another loop after the main while loop to iterate through whichever array was not fully processed and add the rest of its elements.

(b) The if condition on line 7 favors pulling an element from the rightmost array when the two elements are equal to each other. This results in unstable behavior for merge sort, because two equal elements should be added to the merged array in the same order they originally appeared in. To fix this bug, we simply need to change the < on line 7 to be <=.

11. Bonus: One More Question [1 pt]

Now it's your turn – for 1 free bonus point, ask any question in the space below. This can be about the course, your instructor/TAs, the future of human civilization, etc. We'll compile our responses and publish on Piazza after the quarter!

Alternatively, you may list your favorite data structures or algorithm pun. All puns (or attempts thereof) will receive 1 point, though good ones will make the course staff smile.

Solution:

We hope this quarter has *tree*-ted you well!