CSE 373 20au: Exam II Solutions

Note: This exam was originally offered via Gradescope. Free response fields are omitted in this PDF reproduction.

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 8:30 am PST on Friday, December 18.** You may **not** use late days on this exam.
- You may work in multiple sessions, and submit as many times as you'd like (we only grade your last one).
- You may work alone, or in groups of up to 6 (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 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 20au, such as former students.
- During the exam, you may ask clarification questions on Ed or in office hours. 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 posted on Ed.

This problem asks you to affirm you and your group have read all of these policies and those on the website. Failure to fill this question out may result in your exam not being marked for credit.

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. Heap vs AVL Tree [2 pts]

An AVL Tree with the same height as a binary min-heap will have the same number of elements.

◯ ALWAYS ◯ NEVER ◯ SOMETIMES (Need more information)

Solution:

SOMETIMES. This will be true if the AVL tree is also a complete tree, otherwise the number of nodes is not guaranteed to match.

2.2. Heap Size [2 pts]

A binary min-heap of height $h > 0$ (where the last row is completely full) can fit in an array with 2^h entries.

◯ ALWAYS ◯ NEVER ◯ SOMETIMES (Need more information)

Solution:

NEVER. The most nodes that a complete binary tree of height h can hold is $2^{h+1} - 1$ which means we can't use an array of size 2^h to represent any tree of height h.

2.3. Binary Min-Heaps [2 pts]

If two binary min-heaps (implemented using arrays) store the same sequence of values by priority, then their internal arrays must store elements in the same order.

For example, consider these two binary min-heaps `pq1` and `pq2`. They return values in the same order when calling `removeMin`, so this question is asking whether or not their internal arrays store values in the same order.

```
print(pq1.removeMin(), pq2.removeMin())  # prints 1, 1
print(pq1.removeMin(), pq2.removeMin())  # prints 3, 3
print(pq1.removeMin(), pq2.removeMin())  # prints 6, 6
```

◯ ALWAYS ◯ NEVER ◯ SOMETIMES (Need more information)

Solution:

SOMETIMES. Two binary min-heaps with the same internal arrays will return values in the same order, but it's also possible for different arrays to return values in the same order. For example, binary min-heaps don't differentiate between left and right children, so any array that represents a heap with left/right children swapped would return values in the same order.

2.4. Largest Element [2 pts]

The largest element in a binary min-heap will be stored in the last spot in the array representation of the heap.

◯ ALWAYS ◯ NEVER ◯ SOMETIMES (Need more information)

Solution:

SOMETIMES. The largest value will have to be in the last row, which sometimes can be in the last index, but other times can be in many possible indices.

3. Gardening [10 pts]

Learning Objectives: [some objectives omitted to avoid giving away the solution], Identify whether algorithms are considered reductions (LEC17)

Congrats! You have just been hired to maintain UW's plant collection. Your job is to make sure we are able to water all the plants, by connecting them with hoses to water sources.

We have the budget from the state to set up k water sources. Due to the contract UW made, we must construct and use k watering sources, and each one must water at least one plant. The way watering sources work is we can choose to place one on top of a single plant, thus watering that plant.

There are currently n plants housed at UW (and we know $n > k$). For each pair of plants, we know the distance (in meters), between where the plants are currently located on the UW campus. Budget constraints are tight, so we aren't able to relocate the plants. We can easily water k of the n plants by constructing our k watering sources, but the problem is how to water all of the rest.

To water more plants, we can connect plants via hoses that connect them to a plant that has a watering source on it. So for example, if you put a watering source on top of plant A, and connect plant A and B via a hose, plant B will also be watered. Since we already budgeted to construct k watering sources, the cost of making sure all the plants are watered is determined by the length of hose (in meters) needed to connect all the plants to a watering source.

We will assume the mechanics of watering plants follows the rules of ”watering transitivity”: Assume plant A has a watering source on it. If we connect a hose from plant A to plant B, then plant B can also be watered using the source from plant A. If we then connect a hose from plant B to plant C, plant C can also be watered through plant B.

We will also assume there is no restriction of how much water can ”flow” between a plant. If there is a hose between plant B and plant D, and plant B and plant E, both plants D and E can be watered if B is watered. Water can flow in either direction along a hose, so as long as at least one plant is watered in a series of connected plants, all of those connected plants are watered.

3.1. Approach [8 pts]

Describe an algorithm to decide on which plants we should construct our k watering sources on and a plan to connect the plants via hoses, such that the total cost of hoses needed to make sure every plant is watered is minimized.

The input for your algorithm should be a list of n plants and the pairwise distances between them (e.g., the distance between plant A and B is 5m, the distance between plant A and C is 1m, etc.) and the number k of watering sources we need to construct. The distances are symmetric so a distance between plant A and B of 5m, means the distance between plant B and plant A is also 5m. The output of your algorithm should be a plan to decide which plants should have watering sources constructed on top of them, and a plan to decide which plants should be connected by hoses.

For example, suppose we had three plants and wanted to construct 2 watering sources. The input might describe

First Plant	Second Plant	Distance
A	B	5m
A	C	1m
B	C	2m

Then the output for your algorithm should say we should connect A and C by a hose and place a watering source over plant B and then one of Plant A or C. Notice, it doesn’t actually matter which plant you place a watering source on inside a connected component, since the cost is really based on the hose required.

For full credit, your algorithm **must be a reduction**. As a reminder, a reduction requires you to follow the three following steps. You must explicitly, and unambiguously specify each of the following three steps:

(a) Transform the input described above to be used by some pre-written algorithm that you choose for step (b).

(b) Which algorithm discussed in class that you will use to reduce to.

(c) How to transform the output of the algorithm you chose into a solution to the problem stated above.

Note that we are just looking for a description of these steps. We are not looking for you to write code or anything, just unambiguously describe what transformations you would do on the input/output and which algorithm you will use in the problem you are reducing to. It is up to you to describe your solution in enough detail to demonstrate you have solved a problem.

Solution:

(a) Construct a weighted, undirected graph where each plant is a vertex and there is an edge between each pair of vertices with edge weight that is the distance between their respective plants.

(b) Run a Minimum Spanning Tree Algorithm on this graph. Either Kruskal's or Prim's works here. The edges selected are the minimum amount of hose necessary to connect all plants.

(c) Remove the $k-1$ largest edges used in the MST to get k connected components of minimum cost. In the example above, the MST would return the edges (A,C) with cost 1 and (C,B) with cost 2. We remove the heaviest edge to put them in 2 groups, leaving A/C in one group and B on its own. The edges remaining correspond to the hoses we need to connect the plants. Arbitrarily pick any node in each connected component to become the watering source for that connected component.

3.2. Explain Reductions [2 pts]

Explain the benefit of solving a problem like the one above using a reduction. This requires using your own words to explain what a reduction is. (2-3 sentences)

Solution:

Using reductions allows us to solve novel problems using pre-existing algorithms, without having to re-invent the wheel. A reduction just requires transforming the input to our problem to an input that another algorithm can work on, and then transforming the output of that algorithm to the original problem.

4. Graphs [6 pts]

Learning Objectives: Categorize graph data structures based on which properties they exhibit (LEC14), Compare the runtimes of Adjacency Matrix and Adjacency List graph implementations, and select the most appropriate one for a particular problem (LEC14), Define a topological sort and determine whether a given problem could be solved with a topological sort (LEC17)

4.1. Graph Implementation [2 pts]

If you needed to calculate the out-degree of all vertices in a graph, which representation would you prefer (select one):

◯ Adjacency Matrix or ◯ Adjacency List

Solution:

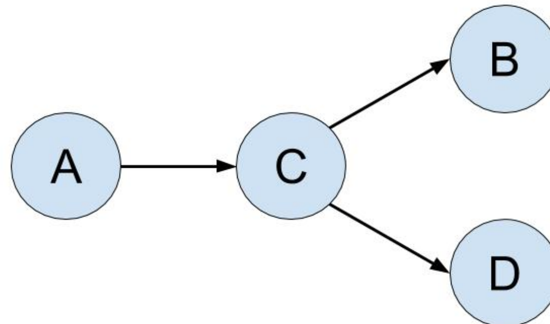
In an adjacency matrix you would need to visit all V locations in the row for that vertex to calculate its out degree (so a total time of $\mathcal{O}(|V|^2)$). In an adjacency list you would only need to visit the index for each vertex in the graph, and compute the length of the list at that vertex ($\mathcal{O}(|V|)$ assuming the lists store the length, $\mathcal{O}(|V|+|E|)$ otherwise).

4.2. Topological Sort [2 pts]

Give an example of a directed graph with 4 nodes, at least one node with in-degree 1, and exactly two different topological orderings. Upload your answer as a PDF or image of the graph and show what the two topological orderings are.

Solution:

One example graph is shown below. The topological orderings of this graph are `A,C,B,D` or `A,C,D,B`.



4.3. Transforming Graphs [2 pts]

We did not formally define **strongly connected** graphs in class. A strongly connected graph is a directed graph where we can reach every node using a directed path in the graph from any start node (this is opposed to the notion we introduced of "weakly connected" graphs where we ignore the directions).

Let G be a connected, undirected, weighted graph. Convert G to a directed graph as follows: replaced every undirected edge (u, v) with directed edges $(u \to v)$ and $(v \to u)$. The resulting graph is **strongly connected**.

◯ TRUE ◯ FALSE

Solution:

TRUE. Since G was originally connected, then it is possible to reach any node B from any starting node A. With the transformation described, we can reach any node B from any start node A since we replace every undirected edge in G with two directed edges, going both ways.

5. Graph Traversals: Code [7 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:

```
interface Vertex {
}

// This represents an undirected edge, meaning to/from don't
// have any different special meanings.
interface Edge {
    public Vertex to();
    public Vertex from();
}

interface Graph {
```

```
        // Gets all of the edges where v is one of the nodes on the edge.
        // Returns a list of edges such that v is always the `from` field.
        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.

```
    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 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. Partition Rooms [7 pts]

Suppose we had an undirected graph, where each vertex corresponded to a person attending a party, and each edge (u, v) represented that u and v were enemies. Your job is to plan a party with all $|V|$ members, but separated into two rooms (Room 1 and Room 2), such that no pair of enemies are in the same room.

Given an input graph and some arbitrary start person (vertex), write a method called `partition` that returns a Map from vertex to room assignment (1 or 2) such that no two nodes that share an edge are assigned the same room. **We will assume the given graph is partitionable**, meaning it's possible to make assignments of room 1 and room 2 such that no two enemies appear in the same room. This means you don't need to do any checks for if the partitioning is not possible.

You may assume that no `Vertex` in the graph is `null`.

Hint: Think carefully about how this enemy relationship relates to a BFS of the graph starting at the given start vertex. What can you say about adjacent levels of the BFS tree?

Hint: Because we let you assume the graph is partitionable into two rooms, you do not need to consider the case if the graph cannot be partitioned into two rooms (e.g., three nodes connected by edges to form a triangle).

Copy the above BFS code and modify it to return the map of room assignments as described. We recommend typing your solution in an IDE (i.e., IntelliJ) and pasting it here afterward.

The method header should be `Map<Vertex, Integer> partition(Graph graph, Vertex start)`.

Solution:

```
Map<Vertex, Integer> partition(Graph graph, Vertex start) {
    Map<Vertex, Integer> rooms = new HashMap<>();
    Queue<Vertex> perimeter = new Queue<>();
    perimeter.add(start);
    rooms.put(start, 1);
    while (!perimeter.isEmpty()) {
        Vertex from = perimeter.remove();
        // Fancy syntax for a 1-line if statement.
        int nextRoom = rooms.get(from) == 1 ? 2 : 1;
        for (Edge edge : graph.edgesFrom(from)) {
            Vertex to = edge.to();
            if (!rooms.contains(to)) {
                perimeter.add(to);
                rooms.put(to, nextRoom);
            }
        }
    }
    return rooms;
}
```

6. Graph Traversals: Analysis [8 pts]

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)

In class, we saw an implementation of Dijkstra's algorithm without a `Set` keeping track of the nodes visited. Our goal in this problem is trying to implement that optimization in the same way with BFS. Consider the following implementation of BFS, called newBFS. The only modification we've made is removing the visited set:

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

6.1. Space Complexity [2 pts]

Is the worst-case space complexity of this implementation *asymptotically better* than the original implementation (see Question 6)?

- ◯ newBFS has a **higher** asymptotic space complexity than BFS
- ◯ newBFS has the **same** asymptotic space complexity as BFS
- ◯ newBFS has a **lower** asymptotic space complexity than BFS

Briefly justify your answer: (1-2 sentences)

Solution:

The space complexity is asymptotically **higher** in the worst case. Removing the visited set saves $\mathcal{O}(|V|)$ space, but the perimeter may now have duplicate vertices in it for each edge that visits a vertex (since there no check for if a vertex has been visited). There $\mathcal{O}(|V|^2)$ edges in a graph, so in the worst case, the perimeter will have $\mathcal{O}(|E| + |V|)$ vertices in it, which is asymptotically more space.

6.2. Finding a Vertex [3 pts]

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

◯ Yes ◯ No

Briefly justify your answer: (1-2 sentences)

Solution:

Yes. "No" is a tempting answer since we ran into a problem with DFS when the graph had a cycle, and getting stuck in an infinite loop. This is actually not a problem for BFS though, since it explores all nodes in a breadth-first manner. It will explore other parts of the graph, even while adding nodes in the cycle back to the queue.

6.3. Dijkstra's Algorithm [3 pts]

This question is not related to newBFS

Consider implementing Dijkstra's algorithm, but instead of using a `PriorityQueue` ADT, we use an **unsorted** list to track the currently unvisited nodes in the perimeter. What would the asymptotic runtime of this alternative Dijkstra's implementation be in the worst case?

Write your answer as an asymptotic runtime and briefly justify your answer (1-2 sentences).

Solution:

$\mathcal{O}(|V|^2 + |E||V|)$. The $\log(|V|)$ terms in Dijkstra's came from the logarithmic operations on a PQ. With an unsorted list, we replace these log operations on the PQ with a $\mathcal{O}(|V|)$ operation to find the next closest vertex, an $\mathcal{O}(|V|)$ operation to update the priority of an existing vertex, and a $\mathcal{O}(1)$ operation to add a new vertex to the perimeter.

7. MSTs [6 pts]

Learning Objectives: Describe the Cut and Cycle properties must be true from the definition of an MST and explain how Prim's Algorithm utilizes the Cut property for its correctness (LEC18), Describe Kruskal's Algorithm, evaluate why it works, and describe why it needs a new ADT (LEC19)

For the following problems, describe how Prim's and Kruskal's utilize the Cut and/or Cycle property for their algorithm correctness.

7.1. Prim's Correctness [3 pts]

Describe how Prim's Algorithm uses the Cut and/or Cycle property for its correctness. Your answer should justify which of the properties it relies on and how the algorithm relies on that assumption for correctness (2-3 sentences).

Solution:

Prim's relies on the Cut Property. Prim's algorithm maintains a known set of nodes that it currently has an "in progress" MST for (the "known set") and it always adds the edge with the smallest weight that goes from the known set to the "unknown set". Choosing this edge is guaranteed to be in the result MST because of the Cut property stating that the minimum edge across the cut (in this example, known/unknown) must being in any result MST.

7.2. Kruskal's Correctness [3 pts]

Describe how Kruskal's Algorithm uses the Cut and/or Cycle property for its correctness. Your answer should justify which of the properties it relies on and how the algorithm relies on that assumption for correctness (2-3 sentences).

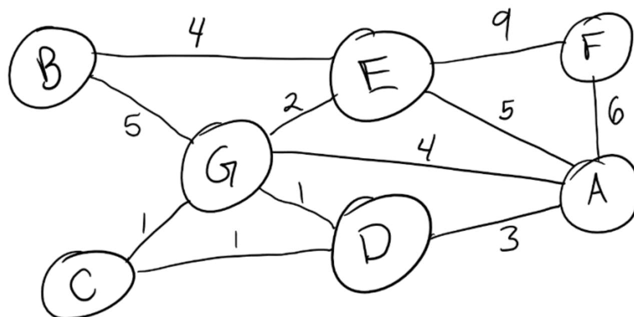
Solution:

Kruskal's relies on the Cycle property for correctness. Kruskal's explores edges from smallest to largest weight, and it will only potentially create a cylce if it is considering an edge that connects nodes in the same connected component, thus causing a cycle in that connected component. Because we explore edges from smallest to largest, we will always consider the heaviest edge that may form a cycle, and we can always throw that edge out due to the Cycle property.

8. Prim's Algorithm [9 pts]

Learning Objectives: Implement Prim's Algorithm and explain how it differs from Dijkstra's (LEC18)

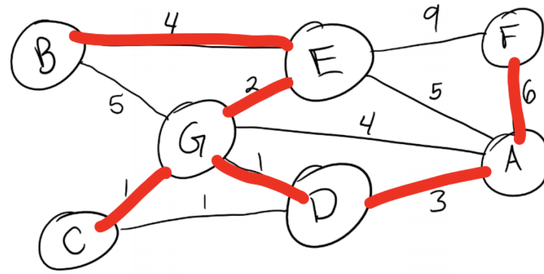
Consider the following graph for the following problems.



8.1. Prim's from G [6 pts]

Find the minimum spanning tree with Prim's algorithm **using vertex G as the starting node**. Upload your answer as a PDF or image of the graph with the edges of the MST highlighted, as well as the table to track the state of Prim's algorithm. You **must show your work** in the table for Prim's algorithm for full credit (show all intermediary steps by crossing out old values). Break ties by choosing the lowest letter first (e.g., if B and C were tied, you would choose B first). *Note: that the next question asks you to recall what order vertices were declared known.*

Solution:



	Cost	Prev	Known?
A	~~∞~~ ~~4~~ 3	~~G~~ D	~~F~~ T
B	~~∞~~ ~~5~~ 4	~~G~~ E	~~F~~ T
C	~~∞~~ 1	G	~~F~~ T
D	~~∞~~ 1	G	~~F~~ T
E	~~∞~~ 2	G	~~F~~ T
F	~~∞~~ ~~9~~ 6	~~E~~ A	~~F~~ T
G	~~∞~~ 0		~~F~~ T

8.2. Known Set [1 pt]

List the order that the nodes in the previous problem are added to the known set.

Solution:

G, C, D, E, A, B, F

8.3. Alternate Start [1 pt]

Pick a node you could start Prim's from in the graph above to get a **different minimum spanning tree** than the one you found in 9.1. Which edge would be in this new tree that is **not** in your tree above? You do not need to draw out the full tree, just specify the start node and the new edge that is added when running Prim's from this start node.

Solution:

Starting node: `C`. Edge: `(C, D)`.

8.4. Negative Weight [1 pt]

Will Prim's starting at vertex G find a correct MST if the weight of edge (A,F) is set to -6? (select one).

◯ YES ◯ NO

Solution:

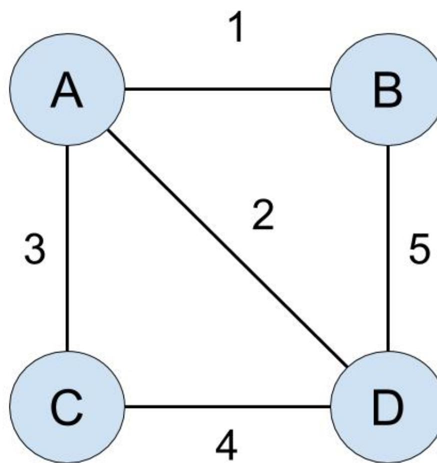
YES. Prim's works with negative weights.

9. Disjoint Sets and Kruskal's [6 pts]

Learning Objectives: Describe Kruskal's Algorithm, evaluate why it works, and describe why it needs a new ADT (LEC19), Implement WeightedQuickUnion and describe why making the change protects against the worst case find runtime (LEC19), implement WeightedQuickUnion using arrays and describe the benefits of doing so (LEC20)

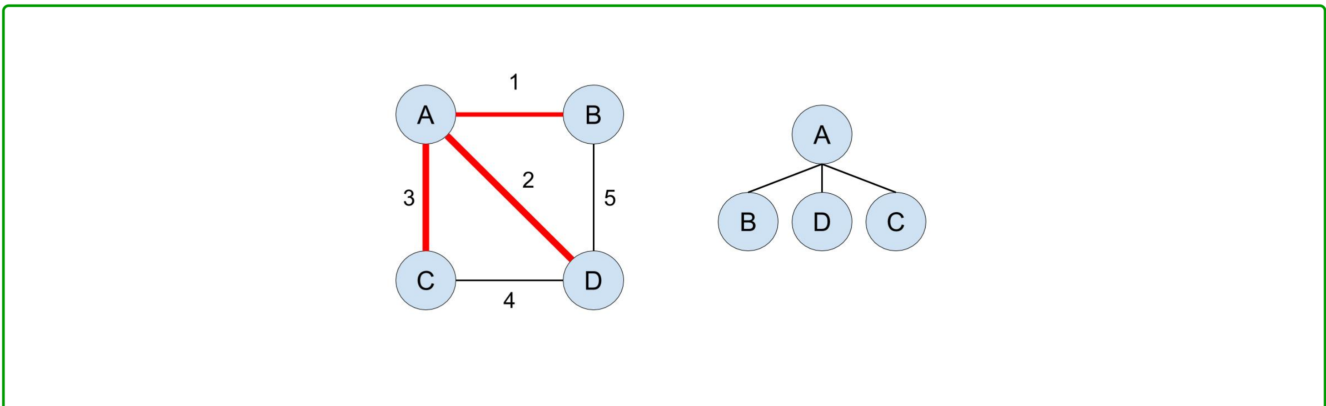
9.1. Kruskal's [4 pts]

Consider the following graph. Show both the MST selected by Kruskal's algorithm and show the state of the internal disjoint set structure used by Kruskal's at the end (as a tree). Assume Kruskal's uses the tree implementation of WeightedQuickUnion, **without** path compression.



Upload your answer as a PDF or image upload. Your answer should show both the MST found and the state of the disjoint set structures used by Kruskal's (as an up-tree).

Solution:

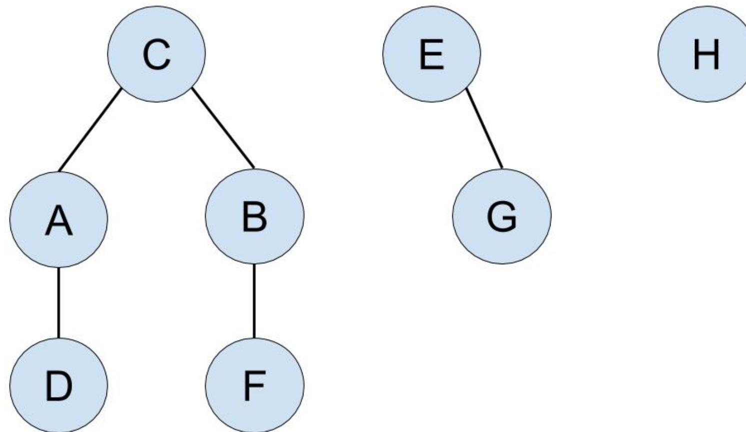


9.2. Disjoint Set Array [2 pts]

Consider the following state of a WeightedQuickUnion disjoint set structure. What is the array representation of this structure, as described in lecture. Write out your answer as an array of integers, with brackets and no spaces between elements (e.g., `[1,2,3]`.

Assume we are using the mapping below from nodes to indices

```
A -> 0
B -> 1
C -> 2
D -> 3
E -> 4
F -> 5
G -> 6
H -> 7
```



Solution:

```
[2,2,-5,0,-2,1,4,-1]
```

10. Sorting [20 pts]

Learning Objectives: Define an ordering relation and stable sort and determine whether a given sorting algorithm is stable (LEC23), Implement Selection Sort and Insertion Sort and compare runtimes and best/worst cases of the two algorithms (LEC23), Implement Heap Sort, describe its runtime, and implement the inplace variant (LEC23), Implement Merge Sort, and derive its runtimes (LEC24), Implement Quick Sort, derive its runtimes, and implement the in-place variant (LEC24)

For all but the last two questions in this section, we will be considering a modification to the in-place QuickSort algorithm described in class.

Suppose we made a new implementation of QuickSort that chose **2 pivots** and separated the partitions into 3 groups. More formally, we would pick two elements as pivots and label the smaller of the two a, and the larger of the two b. We would then partition the array into values

- $... < a$
- $a \leq ... \leq b$
- $b < ...$

We do not show the code of this implementation. Suppose we use a ”median of 4” pivot selection strategy, selecting 4 values from the array and, choosing the middle two values as the two pivots. The 4 values are selected from the ends of the array, and the two indices closest to the middle of the array. Then from those 4 numbers, the two median numbers are decided as the pivots

10.1. Best Case Recurrence [3 pts]

What is the recurrence for this 2-pivot QuickSort in the **best case**?

Follow the recurrence template:

$$T(n) = \begin{cases} \text{BASE_CASE}, & \text{BASE_CONDITION} \\ \text{RECURSIVE_CASE}, & \text{RECURSIVE_CONDITION} \end{cases}$$

On Gradescope, enter the following values into the two textboxes.

- Give BASE_CASE and BASE_CONDITION
- Give RECURSIVE_CASE and RECURSIVE_CONDITION

Solution:

- BASE_CASE and BASE_CONDITION: $c, \text{ if } n \leq 2$
- RECURSIVE_CASE and RECURSIVE_CONDITION: $3T(\frac{n}{3}) + n, \text{ otherwise}$

10.2. Best Case Asymptotic Bound [1 pts]

Using the recurrence you found above, what is the asymptotic runtime of this algorithm in the **best case**?

Solution:

$$\Theta(n \log n)$$

10.3. Worst Case Recurrence [3 pts]

What is the recurrence for this 2-pivot QuickSort in the **worst case**?

Follow the recurrence template:

$$T(n) = \begin{cases} \text{BASE_CASE}, & \text{BASE_CONDITION} \\ \text{RECURSIVE_CASE}, & \text{RECURSIVE_CONDITION} \end{cases}$$

On Gradescope, enter the following values into the two textboxes.

- Give BASE_CASE and BASE_CONDITION
- Give RECURSIVE_CASE and RECURSIVE_CONDITION

Solution:

- BASE_CASE and BASE_CONDITION: $c, \text{ if } n \leq 2$
- RECURSIVE_CASE and RECURSIVE_CONDITION: $T(n-2) + n, \text{ otherwise}$

10.4. Worst Case Asymptotic Bound [1 pts]

Using the recurrence you found above, what is the asymptotic runtime of this algorithm in the best case?

Solution:

$$\Theta(n^2)$$

10.5. Mixed Sorts [3 pts]

Note: This question is unrelated to the previous ones.

Many divide and conquer sorting algorithms (e.g., MergeSort and QuickSort) have a slightly different base case than what we described in class. Instead of stopping at arrays of size 1, they have a base case that stops at some small array (e.g., size 10) and then run an algorithm like Insertion Sort to sort this small array.

Why might implementers of sorting algorithms do this? Explain a potential benefit of this approach and why it can be used for both QuickSort and MergeSort.

Solution:

Although Insertion Sort is asymptotically worse than something like QuickSort or MergeSort, asymptotic runtimes don't apply on small, constant-sized inputs. While QuickSort or Merge Sort are better asymptotically, they may have worse constant factors on small inputs, so a simpler algorithm could run more efficiently on a small input. Having a base case that uses InsertionSort on a small portion of the array can then speed up the whole sorting algorithm because all of the runtime sorting the small subarrays is faster using the simpler algorithm.

10.6. Debugging Insertion Sort [9 pts]

Note: This question is unrelated to the previous ones.

Consider the following **BUGGY** code intended to implement InsertionSort 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. We want to implement InsertionSort such that it has all the properties described in class.

You may assume that the given array does not have any null elements.

```
01 void insertionSort(Element[] data) {
02     for (int i = 1; i < data.length; i++) {
03         // Grab the current element
04         Element value = data[i];
05         int j = i - 1;
06         // Shift over elements in sorted region
07         while (data[j].compareTo(value) >= 0 && j >= 0) {
08             data[j + 1] = data[j];
09             j = j - 1;
10         }
11         data[j] = value;
12     }
13 }
```

The above code contains at least one error and as many as three errors; each error that's present can be fixed in one change to the code. 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 7 is written so that InsertionSort will not be stable. If we shift over equivalent values, we'll put a value later in the array before its tie cases. One way to fix this is to write `data[j] > value` in that condition. There is also a bug in the order of the tests since the `j >= 0` test should come first, but this was less important to us so we did not look for mentions of this.

(b) The current value is stored at the wrong index of the array on line 11. There is an off by one bug here

where it doesn't store the value at the right index (since j will be right before the target location). Should be changed to `data[j + 1] = value`.

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 Ed 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.