I pledge that I have neither given nor received any unauthorized aid on this assignment.

Problem 1 [COMPLETE]

Observations of the execution times in all cases among the selected strategies indicates that the worst cases for executing the Quicksort algorithm ($O(n^2)$) are on sorted sets of data, when the selected pivot is at the extent (min or max value) of the set.

From experimentation, the most effective pivot selection strategies were the midpoint and the median of the start, end, and midpoint values of the data structure. The variance between results on unsorted and sorted sets of data was minimal. This is due to the fact that, in both sorted and unsorted sets, these strategies are more likely to coincide with the better-case selection ($O(n \log n)$) than the worst-case values (a min or max value of an ordered set).

	Pivot Selection Strategy				
	Left	Right	Midpoint	Random	Median
A_{10}	0.0000200272	0.0000231266	0.0000281334	0.0000565847	0.0000238419
A_{100}	0.000315189	0.000272989	0.000282049	0.000634352	0.000262022
A_{200}	0.000636101	0.000571966	0.000596046	0.00131194	0.000585079
A_{300}	0.000967026	0.000821829	0.000961065	0.00202727	0.000841141
A_{400}	0.00136995	0.00123906	0.00129414	0.003012977	0.00120401
A_{500}	0.00158596	0.00159287	0.00162983	0.003496647	0.00161004
A_{600}	0.00226998	0.001827	0.00196195	0.004216033	0.00203514
A_{700}	0.00219393	0.00222182	0.00243902	0.004988033	0.00223088
A_{800}	0.00279117	0.00291491	0.00285292	0.005773623	0.00259495
A_{900}	0.00296688	0.00307894	0.00336003	0.006562393	0.00293398
A_{1000}	0.00365686	0.00330806	0.00366497	0.00742062	0.00328612
A_{2000}	0.00796199	0.00809789	0.00788093	0.015206333	0.00738597
A_{3000}	0.0118921	0.0111051	0.0137901	0.0233167	0.0111001
A_{4000}	0.0157192	0.0156209	0.015806	0.0312957	0.015034
A_{5000}	0.0202911	0.02018	0.0209129	0.039974067	0.0194969
A_{6000}	0.0250561	0.0242341	0.0262821	0.047680333	0.023252
A_{7000}	0.0310249	0.0278671	0.0287609	0.0567484	0.0274282
A_{8000}	0.033514	0.0331879	0.0337598	0.0654623	0.0329268
A_{9000}	0.0398428	0.0378101	0.0420768	0.072516733	0.0376999
A_{10000}	0.043195	0.043612	0.0446351	0.0822087	0.0398562

Table 1: Execution Time on Unsorted Data in Seconds

	Pivot Selection Strategy				
	Left	Right	Midpoint	Random	Median
A_{10}	0.0000319481	0.0000290871	0.0000219345	0.000056982	0.0000238419
A_{100}	0.00134301	0.00101399	0.000248194	0.000610669	0.000259876
A_{200}	0.00458193	0.00378489	0.000496864	0.001232703	0.000558138
A_{300}	0.010479	0.0086062	0.000736952	0.001933417	0.000823021
A_{400}	0.0180161	0.0144148	0.00107694	0.00263802	0.00120091
A_{500}	0.0284009	0.0216951	0.00121689	0.00337402	0.00134492
A_{600}	0.0395989	0.030827	0.00160217	0.004048667	0.00200105
A_{700}	0.0540309	0.041898	0.0021708	0.004736027	0.00217199
A_{800}	0.0691831	0.054198	0.00247002	0.00551136	0.00253892
A_{900}	0.088644	0.0701399	0.00255489	0.00636665	0.00272608
A_{1000}	0.107142	0.0838461	0.00263309	0.006921767	0.00290585
A_{2000}	0.43418	0.344218	0.00581908	0.014558	0.00629306
A_{3000}	0.946405	0.770189	0.00975084	0.022789867	0.0108922
A_{4000}	1.68116	1.4336	0.012074	0.030641	0.013643
A_{5000}	2.66803	2.14788	0.016299	0.038551667	0.0177531
A_{6000}	3.95989	3.0978	0.020705	0.046313067	0.0227902
A_{7000}	5.06046	4.21695	0.0233641	0.053966033	0.0253589
A_{8000}	6.87604	5.72439	0.0264342	0.062006633	0.0282929
A_{9000}	8.59822	7.01017	0.029685	0.0706559	0.0326319
A_{10000}	10.4314	8.85166	0.035387	0.078249367	0.03701

Table 2: Execution Time on Sorted Data in Seconds

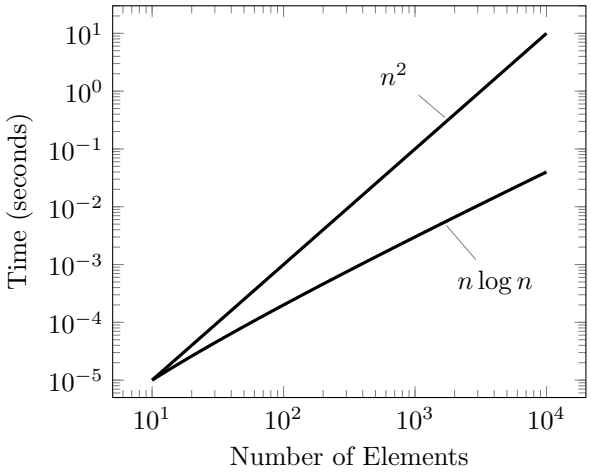


Figure 1: Normalized Efficiency Approximations

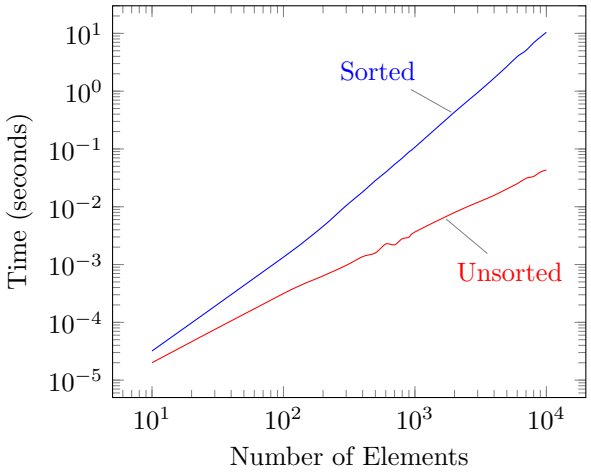


Figure 2: Left-most Pivot Selection

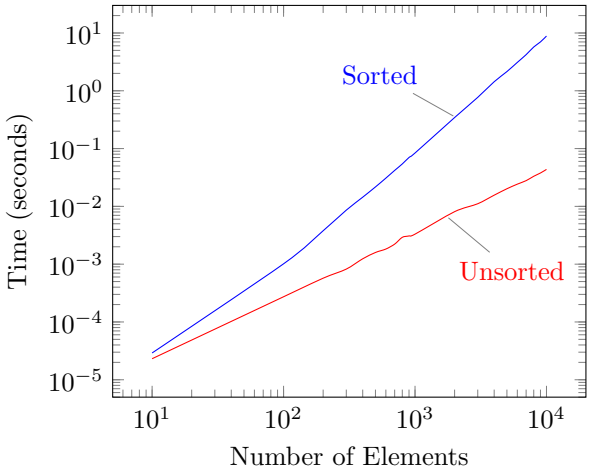


Figure 3: Right-most Pivot Selection

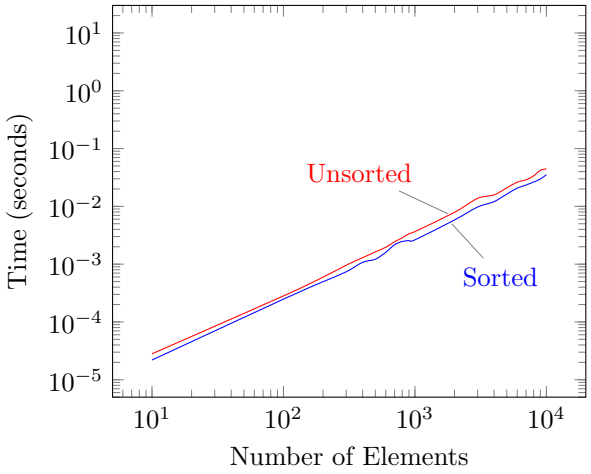


Figure 4: Midpoint Pivot Selection

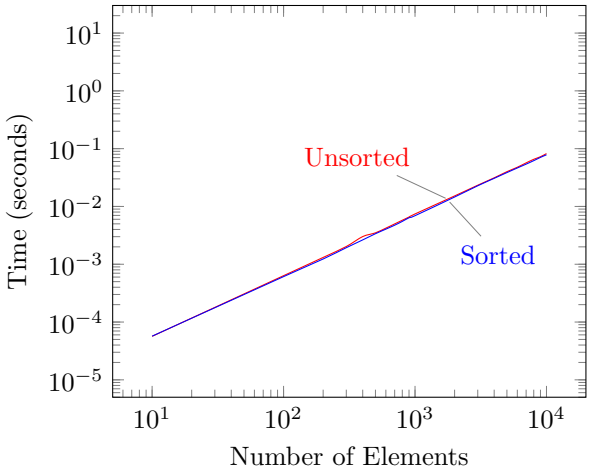


Figure 5: Random Pivot Selection

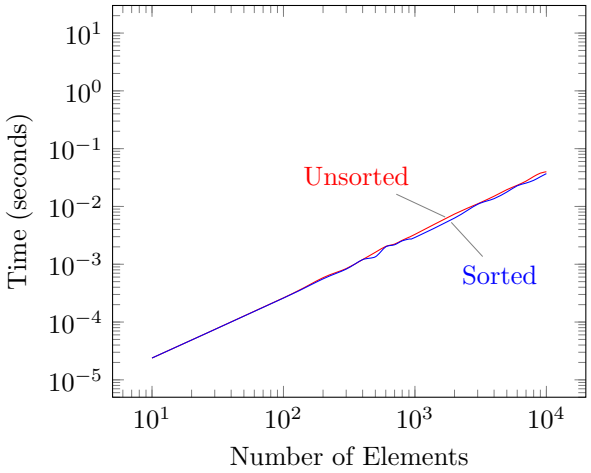


Figure 6: Median Pivot Selection

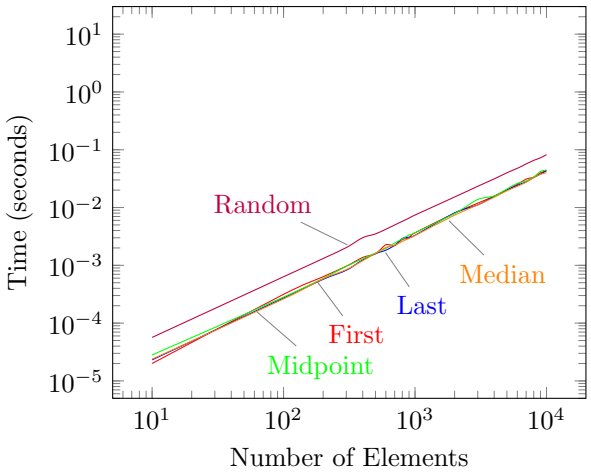


Figure 7: All Pivot Selections on Unsorted Data

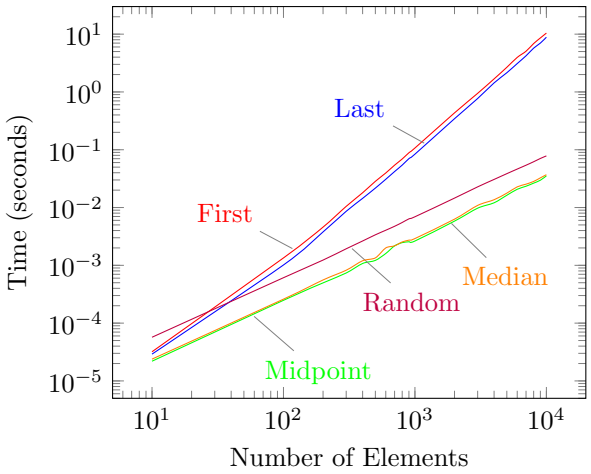


Figure 8: All Pivot Selections on Sorted Data

Problem 2 [COMPLETE]

Greedy 0-1 Knapsack

Process

In this case, the greedy function is maximized for price, packing the knapsack with items in descending order of price, until no more items can be added without exceeding the maximum capacity of the knapsack.

Algorithm 1 Greedily pack a knapsack with items based on price

Inputs: A given capacity (weight limit) W, and S, a set of items, each having a *weight* and *price*.
Outputs: L, a subset of S, consisting of the items to be packed. w, the packed weight. p the packed value.
Complexity: $O(n)$. Each item is computed against one time.

```
1: SortDescending(S, price)          ▷ Sort the items by price, descending
2: w ← 0                             ▷ Weight accumulator
3: p ← 0                             ▷ Price accumulator
4: L ← {}
5: for all s in S do
6:     if w + s.weight ≤ W then      ▷ Item can fit
7:         w ← w + s.weight          ▷ Accumulate the weight
8:         p ← p + s.price           ▷ Accumulate the price
9:         L ← L ∪ {s}               ▷ Merge the item into the output set
10:    end if
11: end for
```

Solution

Given weights of $\{20, 24, 14, 20, 18, 20, 10, 6\}$, prices of $\{15, 9, 27, 12, 36, 12, 9, 12\}$, and a maximum capacity of 80, the optimal 0-1 solution is:

Item		Total	
Weight	Price	Weight	Price
18	36	18	36
14	27	32	63
20	15	52	78
20	12	72	90
6	12	**78**	**102**

Table 3: Greedy 0-1 Knapsack Solution

Greedy Fractional Knapsack

Process

In this case, the greedy function is maximized for value ($weight \div price$), packing the knapsack with items in descending order of value, until no more items can be added without exceeding the maximum capacity of the knapsack. At this point, the fraction of the next item is calculated and added to the knapsack, thereby reaching the maximum capacity while ensuring the maximum possible value.

Algorithm 2 Greedily pack a knapsack with items based on value, including a fractional quantity

Inputs: A given capacity (weight limit) W, and S, a set of items, each having a *weight*, *price*, and *quantity* (default/maximum 1.0).
Outputs: L, a subset of S, consisting of the items to be packed. w, the packed weight. p the packed value.
Complexity: $O(n)$. Each item is computed against one time.

```
1:  SortDescending(S, price)                    ▷ Sort the items by price, descending
2:  w ← 0                                       ▷ Weight accumulator
3:  p ← 0                                       ▷ Price accumulator
4:  L ← {}
5:  for all s in S do
6:      if w + s.weight ≤ W then                ▷ Item can fit
7:          w ← w + s.weight                    ▷ Accumulate the weight
8:          p ← p + s.price                     ▷ Accumulate the price
9:          L ← L ∪ {s}                         ▷ Merge the item into the output set
10:     else
11:         s.quantity ← (W − w) / s.weight     ▷ Adjust the quantity
12:         w ← w + (s.weight * s.quantity)     ▷ Accumulate the modified weight
13:         p ← p + s.price * s.quantity)       ▷ Accumulate the modified price
14:         L ← L ∪ {s}                         ▷ Merge the item into the output set
15:     end if
16: end for
```

Solution

Given weights of $\{20, 24, 14, 20, 18, 20, 10, 6\}$, prices of $\{15, 9, 27, 12, 36, 12, 9, 12\}$, and a maximum capacity of 80, the optimal fractional solution is:

Item				Total	
Weight	Price	Value	Quantity	Weight	Price
18	36	2.00	1.0	18	36
6	12	2.00	1.0	24	48
14	27	1.93	1.0	32	63
10	9	0.90	1.0	52	78
20	15	0.75	1.0	72	90
20	12	0.60	0.6	**80**	**106.2**

Table 4: Greedy Fractional Knapsack Solution

Problem 3 [COMPLETE]

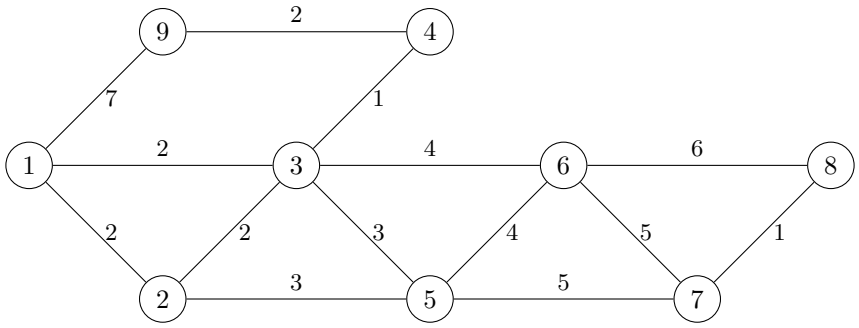
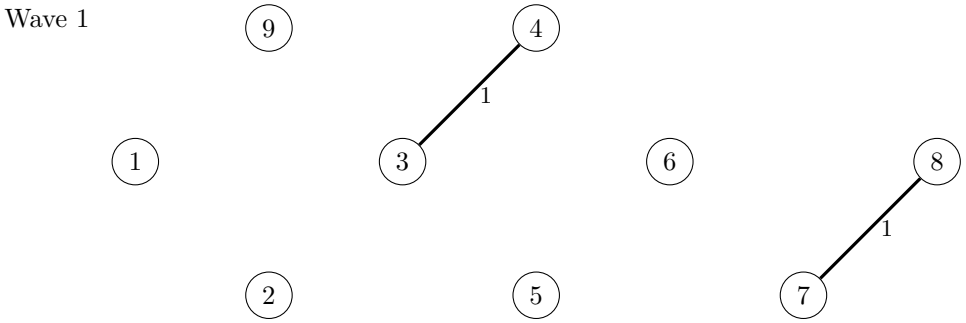
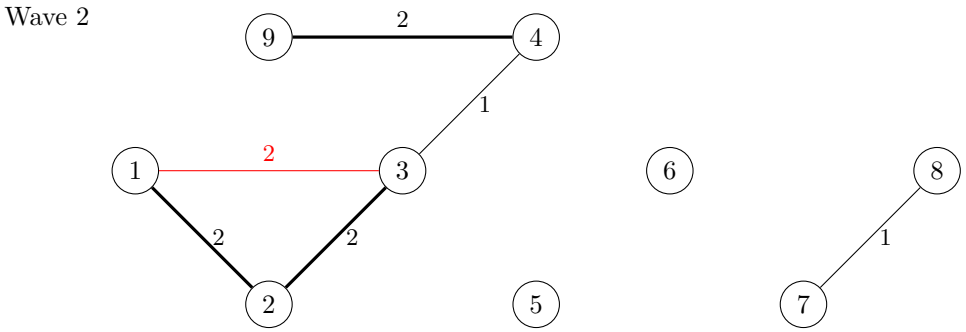


Figure 9: Undirected Graph

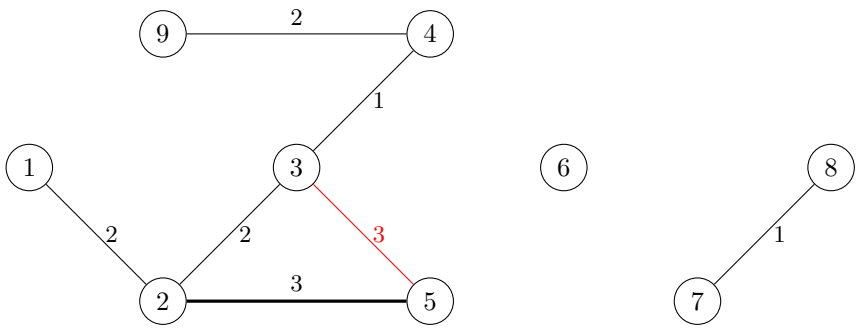
Minimum Spanning Tree

To create the minimum spanning tree (MST), iterate through the edges (in order of weight – Kruskal's Algorithm), taking into consideration only edges with the weight of the current wave (e.g., 1, 2, 3, . . .). The edge is a member of the MST if and only if the edge can be added to the MST without creating a cycle (a closed area within the graph).

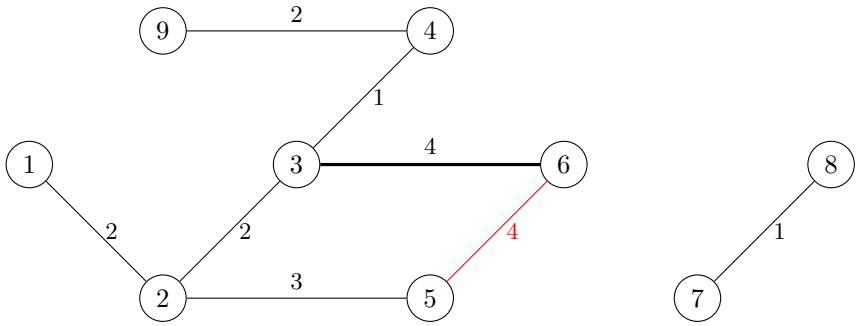




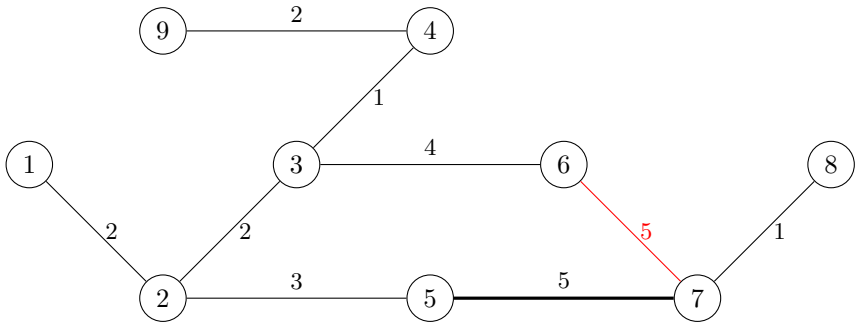
Wave 3



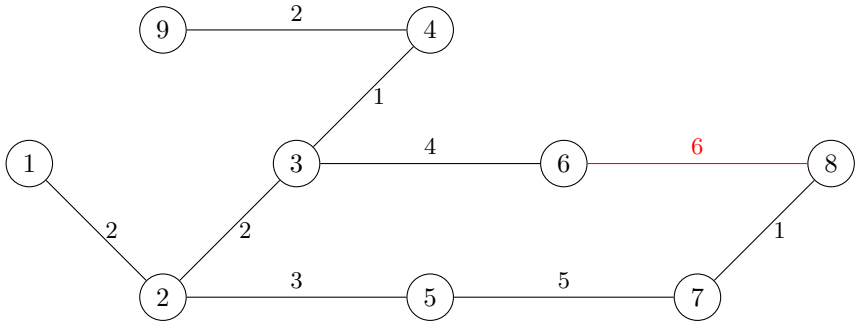
Wave 4



Wave 5



Wave 6



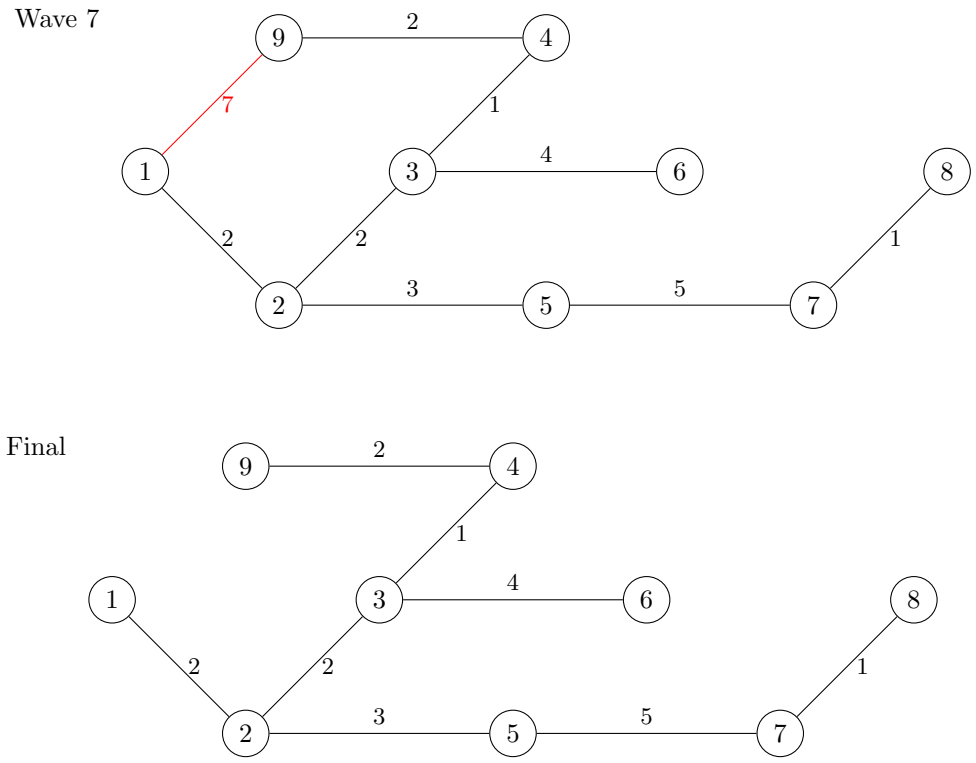


Figure 10: Minimum Spanning Tree

Greedy Single Source Shortest Path

Process

The answer to this problem can be derived using Dijkstra's algorithm. In the following table, u refers to the node of context for each iteration of the loop, while all other values represent the distance from node 1.

u	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)
	0	∞	∞	∞	∞	∞	∞	∞	∞
1	0	**2**	**2**	∞	∞	∞	∞	∞	**7**
2	0	2	2	∞	**5**	∞	∞	∞	7
3	0	2	2	**3**	5	**6**	∞	∞	7
4	0	2	2	3	5	6	∞	∞	**5**
5	0	2	2	3	5	6	**10**	∞	5
9	0	2	2	3	5	6	10	∞	5
6	0	2	2	3	5	6	10	**12**	5
7	0	2	2	3	5	6	10	**11**	5
8	0	2	2	3	5	6	10	11	5

Table 5: Distance Trace of Dijkstra's Single Source Shortest Path Algorithm

Solution

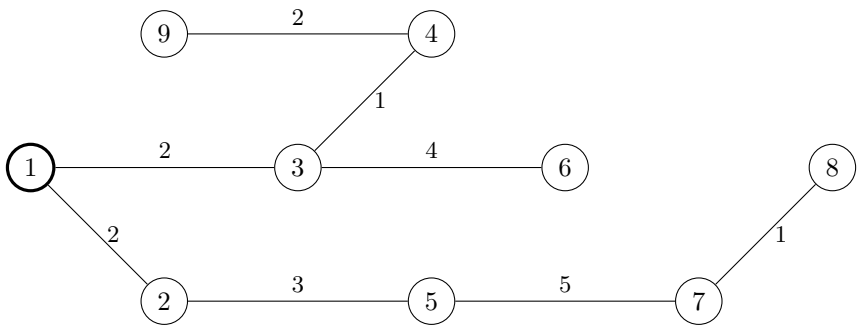


Figure 11: Dijkstra's Single Source Shortest Path Tree

Problem 4 [COMPLETE]

Process

Algorithm 3 Greedily schedule a number of lecture halls for a set of activities

Inputs: S, a set of activities, where each element s has a $start$ and $finish$ property, and $s.start < s.finish$.
Outputs: L, a set of non-conflicting schedule sets.
Complexity: $O(n \log n)$. Each node is computed against one time, without replacement.

```
1: Sort S descending by finish
2: L = {}
3: i = 0
4: while S has elements do
5:     j = 1
6:     for all s in S do
7:         if s_start ≥ j then
8:             L_i = L_i ∪ {s}
9:             j = s_finish
10:            Remove s from S
11:        end if
12:    end for
13:    i = i + 1
14: end while
```

Solution

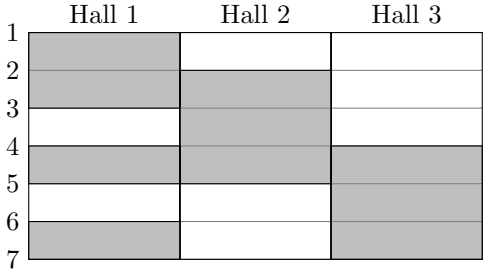


Figure 12: Scheduling Problem Solution

Problem 5 [COMPLETE]

Although both Prim and Kruskal's algorithms are defined using graphs that have positive edge weights, both will correctly function given input edged that have negative weights. Both algorithms iterate through the graph, visiting and declaring edges "safe" if they do not create a cycle in the graph (an enclosed area). Thus, both algorithms will create a MST regardless of input edge weight.