Homework 2

CSE 6140
Computational Science and Engineering

Problem 1

```
for i = 1 to n-1:
    min_index = i
    for j = i+1 to n:
        if A[j] < A[min_index]:
            min_index = j
    swap A[i] with A[min_index]
```

The algorithm needs to run only for the first $n-1$ elements because after $n-1$ iterations, the largest element is already in its correct position, so no further swaps are necessary.

The worst-case running time of selection sort is $O(n^2)$ because there are two nested loops. Even in the best case (where the array is already sorted), the algorithm still checks each pair of elements, so the best-case time complexity is also $O(n^2)$.

Problem 2

(a) The five inversions in the array {2, 3, 8, 6, 1} are:

$$(2,1),(3,1),(8,6),(8,1),(6,1)$$

(b) The array with elements from the set $\{1,2,3,\ldots,n\}$ that has the most inversions is the reverse-sorted array $\{n,\, n-1,\, \ldots,\, 1\}$. It has $\frac{n(n-1)}{2}$ inversions.

(c) The idea is to count inversions while performing the merge operation in a merge sort algorithm. During the merge step, when elements from two halves of the array are merged, if an element from the right half is smaller than an element from the left half, it means that all the remaining elements in the left half form an inversion with that element from the right half (since elements in the left half are larger and appear before those in the right half).

Algorithm Steps:

- Divide: Recursively divide the array into two halves until each subarray contains only one element
- Merge and Count: When merging two halves, count how many elements from the left half are greater than elements from the right half, which is the number of inversions
- Combine: Sum up the inversions from the left half, the right half, and the inversions counted during the merge step

Problem 3

To solve this problem, we can use Dijkstra's algorithm with a priority queue. The key modification is that the priority of each cell is the maximum difference in heights encountered so far on the path to that cell.

rows = r
columns = c

Algorithm:

- Start at the top-left cell (0, 0) with an initial "distance" (effort) of 0. All other cells are initialized with an effort of infinity.
- Use a priority queue where each element is a tuple (currentMaximumEffort, row, column), with currentMaximumEffort representing the maximum height difference encountered along the path so far.
- For each cell, look at its neighboring cells (up, down, left, right). For each neighbor, calculate the maximum difference in heights between the current cell and the neighbor. If this new "maximum difference" is less than the previously recorded value for the neighbor, update the neighbor's value and push it into the priority queue.
- Repeat the process until the bottom-right cell is reached with the minimum possible effort.

Correctness:

- The algorithm uses a greedy approach similar to Dijkstra's algorithm. It always expands the node with the smallest current maximum effort, ensuring that the first time we reach the bottom-right corner, we have found the minimum possible maximum effort.
- Once we find the minimum effort to reach a certain cell, any subsequent path that reaches that cell must have a higher or equal effort, ensuring that the effort values we compute are optimal.
- The algorithm terminates when the bottom-right cell is dequeued from the priority queue, and the value associated with that cell will be the minimum possible effort to reach it from the top-left corner. This guarantees correctness.

Running time: $\mathcal{O}(r \cdot c \cdot \log(r \cdot c))$

Problem 4

To solve this problem, we can use Kruskal's algorithm for finding a Minimum Spanning Tree.

Algorithm:

- First, sort all the edges (roads) by their cost in non-decreasing order.
- Use a Union Find data structure to help detect cycles and connect components efficiently. Each city will initially be in its own set. The algorithm keeps track of which cities are connected by using the Union-Find to merge sets when a road is added.
- Process the sorted edges one by one. For each edge, check if the two cities it connects are in different sets (i.e., they are not yet connected). If they are in different sets, add this edge (road) to the MST and merge the two sets using the Union-Find structure. Continue this process until you have added $n-1$ edges (for n cities, an MST has exactly $n-1$ edges).
- Repeat the process until the bottom-right cell is reached with the minimum possible effort.

Correctness:

- Kruskal's algorithm is a greedy algorithm. It always adds the smallest possible edge that doesn't form a cycle, ensuring that the current partial solution remains a valid MST.
- By using the Union-Find data structure, the algorithm efficiently checks whether adding an edge would form a cycle. If adding an edge does not form a cycle, it is added to the MST.
- Kruskal's algorithm guarantees that the resulting tree is a spanning tree and that the sum of the edge weights is minimized because it always picks the smallest available edge that maintains the spanning tree property.

Running time:
Sorting all the edges takes $\mathcal{O}(E \log E)$, where E is the number of edges. Each union or find operation is extremely slow-growing, so we can consider it $\mathcal{O}(1)$. Since we perform $\mathcal{O}(E)$ operations, the total time for Union-Find is $\mathcal{O}(E)$.

Thus, the total running time of Kruskal's algorithm is $\mathcal{O}(E \log E)$, where E is the number of edges in the graph. Since the number of edges E can be at most $\frac{n(n-1)}{2}$ in a fully connected graph, this simplifies to $\mathcal{O}(n^2 \log n)$ in the worst case.

Problem 5

We can prove the correctness of this greedy algorithm using a staying ahead argument similar to that used for proving optimality in the Interval Scheduling problem.

- Assume the Greedy Solution Uses k Trucks: Let the greedy algorithm use k trucks to pack all the boxes. Each truck is filled with as many boxes as possible without exceeding the weight limit W.
- Consider Any Alternative Solution: Suppose there is some alternative algorithm or strategy that uses fewer than k trucks to transport the boxes. Let's denote the number of trucks used by this alternative solution as k', where $k' < k$.
- Compare Truck Utilization: In the greedy algorithm, each truck is filled as much as possible before sending it off. This means the remaining capacity of any truck at the time it is sent away is less than the weight of the next box that cannot fit. If another solution uses fewer trucks, it implies that at least one of the trucks in the alternative solution must hold more weight than in the greedy solution, because the greedy algorithm exhausts the capacity of each truck as much as possible.
- Staying Ahead Argument: Suppose the greedy algorithm packs up to box i and has already sent off fewer or the same number of trucks as the alternative solution. If the greedy algorithm is sending more trucks at any point, it must be because a new box doesn't fit in the current truck, meaning that every truck in the greedy solution is packed as fully as possible. Therefore, any alternative solution that claims to use fewer trucks must have at least one truck that was not fully packed (i.e., it had remaining capacity but could not fit the next box), which contradicts the assumption that fewer trucks are used.
- Contradiction: The contradiction arises from the fact that any solution using fewer trucks must leave some trucks underutilized. However, since the greedy algorithm always tries to pack the current truck as much as possible before sending it, no alternative can use fewer trucks while maintaining the weight limits. Hence, no alternative solution can stay ahead of the greedy algorithm because the greedy algorithm always "stays ahead" in terms of packing efficiency.
- Conclusion: The greedy algorithm minimizes the number of trucks by ensuring that each truck carries the maximum number of boxes it can hold without exceeding the weight limit W. Any alternative strategy that claims to use fewer trucks will either violate the weight limit or leave some trucks underutilized, which is suboptimal.

Running time:

- For each box, we check whether it fits into the current truck. This check is an $O(1)$ operation (simple weight comparison).
- Since we process each box exactly once and perform a constant amount of work per box, the total time complexity is $O(n)$, where n is the number of boxes.

Thus, the greedy algorithm runs in linear time, $O(n)$, and is optimal in minimizing the number of trucks used.