HW2- Solutions

1.

```
SELECTION-SORT(A, n)
 for i = 1 to n − 1
     smallest = i
     for j = i + 1 to n
         if A[j] < A[smallest]
             smallest = j
     exchange A[i] with A[smallest]
```

Running time is $O(n^2)$ in all cases.

2 .

a. The inversions are $(1, 5), (2, 5), (3, 4), (3, 5), (4, 5)$. (Remember that inversions are specified by indices rather than by the values in the array.)

b. The array with elements drawn from $\{1, 2, \ldots, n\}$ with the most inversions is $\langle n, n-1, n-2, \ldots, 2, 1\rangle$. For all $1 \le i < j \le n$, there is an inversion (i, j). The number of such inversions is $\binom{n}{2} = n(n-1)/2$.

c. Follow the hint and modify mergesort.

To start, let us define a ***merge-inversion*** as a situation within the execution of merge sort in which the MERGE procedure, after copying $A[p:q]$ to L and $A[q+1:r]$ to R, has values x in L and y in R such that $x > y$. Consider an inversion (i, j), and let $x = A[i]$ and $y = A[j]$, so that $i < j$ and $x > y$. We claim that if we were to run merge sort, there would be exactly one merge-inversion involving x and y. To see why, observe that the only way in which array elements change their positions is within the MERGE procedure. Moreover, since MERGE keeps elements within L in the same relative order to each other, and correspondingly for R, the only way in which two elements can change their ordering relative to each other is for the greater one to appear in L and the lesser one to appear in R. Thus, there is at least one merge-inversion involving x and y. To see that there is exactly one such merge-inversion, observe that after any call of MERGE that involves both x and y, they are in the same sorted subarray and will therefore both appear in L or both appear in R in any given call thereafter. Thus, we have proven the claim.

We have shown that every inversion implies one merge-inversion. In fact, the correspondence between inversions and merge-inversions is one-to-one. Suppose we have a merge-inversion involving values x and y, where x originally was $A[i]$ and y was originally $A[j]$. Since we have a merge-inversion, $x > y$. And since x is in L and y is in R, x must be within a subarray preceding the subarray containing y. Therefore x started out in a position i preceding y's original position j, and so (i, j) is an inversion.

Having shown a one-to-one correspondence between inversions and merge-inversions, it suffices for us to count merge-inversions.

Consider a merge-inversion involving y in R. Let z be the smallest value in L that is greater than y. At some point during the merging process, z and y will be the "exposed" values in L and R, i.e., we will have $z = L[i]$ and $y = R[j]$ in line 13 of MERGE. At that time, there will be merge-inversions involving y and $L[i], L[i+1], L[i+2], \ldots, L[n_L - 1]$, and these $n_L - i$ merge-inversions will be the only ones involving y. Therefore, we need to detect the first time that z and y become exposed during the MERGE procedure and add the value of $n_L - i$ at that time to the total count of merge-inversions.

The following pseudocode, modeled on merge sort, works as we have just described. It also sorts the array A.

```
MERGE-INVERSIONS$(A, p, q, r)$
  $n_L = q - p + 1$
  $n_R = r - q$
  let $L[0 : n_L - 1]$ and $R[0 : n_R - 1]$ be new arrays
  for $i = 0$ to $n_L - 1$
      $L[i] = A[p + i - 1]$
  for $j = 0$ to $n_R - 1$
      $R[j] = A[q + j]$
  $i = 0$
  $j = 0$
  $k = p$
  $inversions = 0$
  while $i < n_L$ and $j < n_R$
      if $L[i] \leq R[j]$
          $inversions = inversions + n_L - i$
          $A[k] = L[i]$
          $i = i + 1$
      else $A[k] = R[j]$
          $j = j + 1$
      $k = k + 1$
  while $i < n_L$
      $A[k] = L[i]$
      $i = i + 1$
      $k = k + 1$
  while $j < n_R$
      $A[k] = R[j]$
      $j = j + 1$
      $k = k + 1$
  return $inversions$
```

```
COUNT-INVERSIONS$(A, p, r)$
  $inversions = 0$
  if $p < r$
      $q = \lfloor (p + r)/2 \rfloor$
      $inversions = inversions +$ COUNT-INVERSIONS$(A, p, q)$
      $inversions = inversions +$ COUNT-INVERSIONS$(A, q + 1, r)$
      $inversions = inversions +$ MERGE-INVERSIONS$(A, p, q, r)$
  return $inversions$
```

The initial call is COUNT-INVERSIONS$(A, 1, n)$.

In MERGE-INVERSIONS, whenever $R[j]$ is exposed and a value greater than $R[j]$ becomes exposed in the L array, we increase *inversions* by the number of remaining elements in L. Then because $R[j + 1]$ becomes exposed, $R[j]$ can never be exposed again never be exposed again.

Since we have added only a constant amount of additional work to each procedure call and to each iteration of the last **for** loop of the merging procedure, the total running time of the above pseudocode is the same as for merge sort: $\Theta(n \lg n)$.

3. Modified Dijkstra's algorithm:

```
class Solution:
    def minimumEffortPath(self, heights: List[List[int]]) -> int:
        row = len(heights)
        col = len(heights[0])
        difference_matrix = [[math.inf]*col for _ in range(row)]
        difference_matrix[0][0] = 0
        visited = [[False]*col for _ in range(row)]
        queue = [(0, 0, 0)]  # difference, x, y
        while queue:
            difference, x, y = heapq.heappop(queue)
            visited[x][y] = True
            for dx, dy in [[0, 1], [1, 0], [0, -1], [-1, 0]]:
                adjacent_x = x + dx
                adjacent_y = y + dy
                if 0 <= adjacent_x < row and 0 <= adjacent_y < col and not visited[
                        adjacent_x][adjacent_y]:
                    current_difference = abs(
                        heights[adjacent_x][adjacent_y]-heights[x][y])
                    max_difference = max(
                        current_difference, difference_matrix[x][y])
                    if difference_matrix[adjacent_x][adjacent_y] > max_difference:
                        difference_matrix[adjacent_x][adjacent_y] = max_difference
                        heapq.heappush(
                            queue, (max_difference, adjacent_x, adjacent_y))
        return difference_matrix[-1][-1]
```

4. The n cities and potential roads form an undirected weighted graph. The problem is to find a minimum spanning tree of the graph. We can use Prim's or Kruskal's Algorithm to find the solution.

5.

Say n boxes arrive in the order $b_1, \ldots, b_n$. Say each box b_i has a positive weight w_i, and the maximum weight each truck can carry is W. To pack the boxes into N trucks *preserving the order* is to assign each box to one of the trucks $1, \ldots, N$ so that:

- No truck is overloaded: the total weight of all boxes in each truck is less or equal to W.
- The order of arrivals is preserved: if the box b_i is sent before the box b_j (i.e. box b_i is assigned to truck x, box b_j is assigned to truck y, and $x < y$) then it must be the case that b_i has arrived to the company earlier than b_j (i.e. $i < j$).

We prove that the greedy algorithm uses the fewest possible trucks by showing that it "stays ahead" of any other solution. Specifically, we consider any other solution and show the following. If the greedy algorithm fits boxes $b_1, b_2, \ldots, b_j$ into the first k trucks, and the other solution fits $b_1, \ldots, b_i$ into the first k trucks, then $i \leq j$. Note that this implies the optimality of the greedy algorithm, by setting k to be the number of trucks used by the greedy algorithm.

We will prove this claim by induction on k. The case $k = 1$ is clear; the greedy algorithm fits as many boxes as possible into the first truck. Now, assuming it holds for $k - 1$: the greedy algorithm fits j' boxes into the first $k - 1$, and the other solution fits $i' \leq j'$. Now, for the k^{th} truck, the alternate solution packs in $b_{i'+1}, \ldots, b_i$. Thus, since $j' \geq i'$, the greedy algorithm is able at least to fit all the boxes $b_{j'+1}, \ldots, b_i$ into the k^{th} truck, and it can potentially fit more. This completes the induction step, the proof of the claim, and hence the proof of optimality of the greedy algorithm.