PROBLEM 1:

Consider the problem of making change for n cents using the fewest number of coins. Assume that each coin's value is an integer.

a. Describe a greedy algorithm to make change consisting of quarters, dimes, nickels, and pennies. Prove that your algorithm yields an optimal solution.

The greedy algorithm for making change involves repeatedly selecting the largest possible coin denomination that can be used without exceeding the remaining amount of change needed.

ALGORITHM:

Algorithm 1 Make-Change(Amount)

```
Count ← 0
while Amount > 0 do
    if Amount ≥ 25 then
        Count ← Count + (Amount / 25)
        Amount ← Amount % 25
    else if Amount ≥ 10 then
        Count ← Count + (Amount / 10)
        Amount ← Amount % 10
    else if Amount ≥ 5 then
        Count ← Count + (Amount / 5)
        Amount ← Amount % 5
    else
        Count ← Count + Amount
        Amount ← 0
    end if
end while
return Count
```

To prove that this algorithm is correct, we need to show that it always produces the correct output for any input amount of change. We can prove this by induction on the value of the input amount.

Base Case:
When the input amount is 0, the algorithm correctly returns a count of 0, since no change needs to be made.

Inductive Hypothesis:
Assume that the algorithm correctly calculates the minimum number of coins needed to make change for any input amount less than k, where k is some positive integer.

Inductive Step:
Consider an input amount of k. Let C be the correct number of coins needed to make change for k, and let C' be the number of coins returned by the algorithm. We need to show that C = C'. Since the algorithm uses a greedy approach, it always chooses the largest possible coin denomination that can be used without exceeding the remaining amount of change needed. This ensures that the algorithm always uses the minimum number of coins possible. Suppose, for the sake of contradiction, that $C > C'$. This means that the algorithm did not use enough coins to make change for k, which implies that there must be a coin denomination that the algorithm did not choose, but that is necessary to make change for k. However, this contradicts the fact that the algorithm uses a greedy approach, which always chooses the largest possible coin denomination. Therefore, C cannot be greater than C'. Similarly, suppose that $C < C'$. This means that the algorithm used too many coins to make change for k, which implies that there must be a way to make change for k using fewer coins than C. However, this contradicts the inductive hypothesis, which assumes that the algorithm correctly calculates the minimum number of coins needed for any input amount less than k. Therefore, C cannot be less than C'.

Therefore, we have shown that C = C', which proves that the algorithm correctly calculates the minimum number of coins needed to make change for any input amount. Therefore, the algorithm is correct. Inductive Step:

b. Suppose that the available coins are in the denominations that are powers of c, i.e., The denominations are c^0, c^1, ..., c^k for some integers $c > 1$ and $k \geq 1$. Show that the greedy algorithm always yields an optimal solution.

We can prove that the greedy algorithm always yields an optimal solution when the available coins are in the denominations that are powers of c, where $c > 1$ and $k \geq 1$, using a proof by contradiction.

Assume that there exists an input amount of change n for which the greedy algorithm does not yield an optimal solution. Let C_{g} be the set of coins returned by the greedy algorithm, and let C_{opt} be the set of coins that make up the optimal solution.

Since C_{g} is not optimal, there must be at least one coin denomination d in C_{opt} that is not in C_{g}. Without loss of generality, we can assume that d is the smallest denomination in C_{opt} that is not in C_{g}.

Let m be the largest integer such that $d > c^m$. Since the denominations are powers of c, we know that d is of the form $c^m \cdot d'$, where d' is a positive integer less than c. Therefore, d' is the largest denomination less than or equal to d that is not in C_{g}.

Let r be the remaining amount of change after selecting the coins in C_{g}. Since the greedy algorithm selects the largest denomination less than or equal to the remaining amount of change, it follows that the largest coin denomination selected by the greedy algorithm for r is less than or equal to d'. Therefore, the greedy algorithm would have selected d' if it were available.

However, since d' is not in C_{g}, it must be the case that the remaining amount of change after selecting the coins in C_{g} is less than d'. Let r' be the remaining amount of change after selecting the coins in C_{g} except for d'. Since d' is the largest denomination less than or equal to d that is not in C_{g}, it follows that $r' \geq 0$.

Now, consider the set of coins C'_{opt} obtained by replacing d with d' and selecting the optimal set of coins for r'. Since C_{opt} is an optimal set of coins for n, and d is the smallest denomination in C_{opt} that is not in C_{g}, it follows that the total number of coins in C'_{opt} is at most the total number of coins in C_{opt}.

However, since the denominations are powers of c, we know that d' is the largest denomination less than or equal to d that is not in C_{g}. Therefore, C'_{opt} is a valid set of coins for n, and it uses at most k coin denominations, since all the denominations used by C_{g} are also used by C'_{opt}.

Since the total number of coins in C'_{opt} is at most the total number of coins in C_{opt}, and C_{opt} is optimal, it follows that C'_{opt} is also optimal. However, since the greedy algorithm would have selected d' if it were available, it follows that the number of coins in C_g is at least the number of coins in C'_{opt}, which is a contradiction. Therefore, our assumption that the greedy algorithm does not yield an optimal solution is false, and the greedy algorithm always yields an optimal solution when the available coins are in the denominations that are powers of c.

c. Give a set of coin denominations for which the greedy algorithm does not yield an optimal solution. Your set should include a penny so that there is a solution for every value of n

An example of a set of coin denominations for which the greedy algorithm does not always yield an optimal solution is:
$\{1, 2, 3\}$
The greedy algorithm fails for this set, consider the case where n = 6. The greedy algorithm would select 1 coin of 4 and two coins of denomination 1, for a total of 3 coins. However, the optimal solution in this case is to use two coins of denomination 3, for a total of 2 coins. This example shows that the greedy algorithm, which always selects the largest coin denomination that fits into the remaining value, does not necessarily lead to the fewest number of coins needed to make change for a given amount.

d. Give an O(nk) time algorithm that makes change for any set of k different coin denominations, assuming that one of the coins is a penny.

The O(nk) dynamic programming algorithm to make change for any set of k different coin denominations, is as follows,

Algorithm 2 Make-change for any set of k different coin denominations

```
1: Let C[0...n] be an array of length n+1 to represent the minimum number of coins needed to make change for
   every value from 0 to n. Initialize C[0] to 0 and C[i] to infinity for all other i.
2: for $j \leftarrow 1$ to $k$ do
3:     for $i \leftarrow j$ to $n$ do
4:         $C[i] \leftarrow \min\{C[i], C[i-j]+1\}$
5:     end for
6: end for
7: return C[n]
```

The algorithm works as follows:

- At first, the minimum number of coins needed to make change for 0 cents is zero, and for all other values it is infinity.
- For each coin denomination j, we consider all values from j to n. If we can make change for $i-j$ cents using fewer coins than we previously needed to make change for i cents, we update $C[i]$ to reflect this. Specifically, we add one coin of denomination j to the minimum number of coins needed to make change for $i-j$ cents.
- After iterating through all denominations, $C[n]$ will contain the minimum number of coins needed to make change for n cents.

The time complexity of this algorithm is O(nk), since we need to consider all possible values from j to n for each coin denomination, and we do this for k different coin denominations. Note that we assume one of the coins is a penny, which guarantees that there is always a solution for any positive integer value of n.

PROBLEM 2:

Algorithm 3 Count-Stops(d, p):

```
1: $lastStation \leftarrow 1$
2: $numStops \leftarrow 1$
3: for $i \leftarrow 2$ to $n$ do
4:     if $d[i] - d[lastStation] > p$ then
5:         $numStops \leftarrow numStops + 1$
6:         $lastStation \leftarrow i - 1$
7:     end if
8: end for
9: return $numStops$
```

We are assuming that we stopped at the first gas station and filled the tank. This algorithm works by iterating over all gas stations along the route from the second gas station to the last gas station. It checks if the distance between the current gas station and the last gas station where a stop was made is greater than the maximum distance the car can travel (p). If the distance is greater than p, then a stop needs to be made at the previous gas station than the current gas station, so the variable ”NumStops” is incremented and the previous gas station is marked as the last gas station where a stop was made.
The time complexity of this algorithm is O(n), where n is the number of gas stations along the route. This is because the algorithm iterates over all gas stations along the route only once. And all other operations are just assigning and incrementing i.e, takes constant time. Therefore, this algorithm is efficient and suitable for large-scale road trips

To prove the correctness of this algorithm, we need to show that it always returns the minimum number of stops required to travel the entire route. We can do this by contradiction.

Suppose that there exists a better solution, that is, a solution that requires fewer stops than the one returned by the algorithm. Let k be the number of stops in this better solution. Since k is less than the number of stops returned by the algorithm, we know that k is also less than or equal to $n-1$, since there are only $n-1$ distances between gas stations along the route.

Now consider the first stop in the better solution. Let d_i be the gas station where the car stops. Since this is the first stop in the better solution, we know that $d_i - d_1 > p$, otherwise the car could have made it to the second gas station without stopping. We also know that $d_i - d_1 \leq p$, since the algorithm would have stopped at the second gas station if $d_i - d_1 > p$.

Next, consider the second stop in the better solution. Let d_j be the gas station where the car stops. Since this is the second stop in the better solution, we know that $d_j - d_i > p$, otherwise the car could have made it to the third gas station without stopping. We also know that $d_j - d_i \leq p$, since the algorithm would have stopped at the third gas station if $d_j - d_i > p$.

Continuing in this way, we can see that the distances between adjacent stops in the better solution are all greater than p and less than or equal to $2p$. However, since there are only $n-1$ distances between gas stations along the route, there must be at least one pair of adjacent gas stations that are more than $2p$ miles apart. This means that the car will run out of gas before reaching the second gas station in this pair, and so the better solution cannot be optimal.

Therefore, the algorithm Count-Stops returns the minimum number of stops required to travel the entire route, and is therefore correct.

PROBLEM 3:

Algorithm 4 Count-Stops(d, p):

```
1: Sort the heights of the skiers in descending order.
2: Sort the lengths of the skis in descending order.
3: Now, match the skis with skiers such that the first ski is given to the first skier as the first ski is the longest
   and the first skier has the maximum height.
4: Continue and assign the second ski to the second person and so on until the last ski is given to the last skier.
```

Proof by contradiction:

Suppose there exists an optimal solution where the skis are assigned to skiers in a different order than our algorithm. Let's consider the first pair of skier and ski that differ in our algorithm and the optimal solution.

Case 1:

The skier in the optimal solution is taller than the skier in our algorithm, but is assigned a shorter ski. In this case, we can swap the skis between the two skiers without increasing the absolute difference between the heights and lengths. This is because the skier in our algorithm is shorter than the skier in the optimal solution, so the difference in height with the shorter ski is smaller for the skier in our algorithm. The difference in length with the longer ski is also smaller for the skier in our algorithm. Therefore, swapping the skis will result in a better solution.

Case 2:

The skier in the optimal solution is shorter than the skier in our algorithm, but is assigned a longer ski. In this case, we can again swap the skis between the two skiers without increasing the absolute difference between the heights and lengths. This is because the skier in our algorithm is taller than the skier in the optimal solution, so the difference in height with the longer ski is smaller for the skier in our algorithm. The difference in length with the shorter ski is also smaller for the skier in our algorithm. Therefore, swapping the skis will result in a better solution. Therefore, in both cases, we can obtain a better solution by swapping the skis between the two skiers. This contradicts the assumption that the original solution was optimal. Hence, our algorithm is the optimal solution to the problem.

The Time complexity of the algorithm is O(nlogn) where n is the number of skiers or skis. It is taken when we sort the arrays. Both of the arrays are sorted first which will take nlogn time and the loop after that will take n time.

$T(n) : O(n \log n + n \log n + n) \Rightarrow O(n \log n)$

PROBLEM 4:

The problem can be solved efficiently using dynamic programming. We can define a 2D array dp with the same dimensions as the input matrix F, where dp[i][j] represents the maximum amount of gold the miner can collect starting from cell F[i][j] and moving only to the right, diagonally up towards the right, or diagonally down towards the right.
We can initialize the first column of dp with the same values as the first column of F, since the miner can only move to the right from those cells. Then, for each subsequent column of dp, we can calculate the maximum amount of gold the miner can collect starting from each cell by considering the three possible directions of movement and selecting the cell that yields the maximum amount of gold.

Algorithm 5 Collecting Gold:

```
Let n, m be the dimensions of the input matrix F
Initialize dp[n][m] with all values set to 0;
for $i$ from 0 to $n-1$ do
    $dp[i][0] = F[i][0]$
end for
for $j$ from 1 to $m-1$ do
    for $i$ from 0 to $n-1$ do
        if $i > 0$ then
            $dp[i][j] = \max(dp[i][j], dp[i-1][j-1] + F[i][j])$
        end if
        $dp[i][j] = \max(dp[i][j], dp[i][j-1] + F[i][j])$
        if $i < n-1$ then
            $dp[i][j] = \max(dp[i][j], dp[i+1][j-1] + F[i][j])$
        end if
    end for
end for
$max\_gold = dp[0][m-1]$
for $i$ from 1 to $n-1$ do
    $max\_gold = \max(max\_gold, dp[i][m-1])$
end for
return $max\_gold$
```

Let's assume that there exists a better solution that our algorithm did not consider. Let S be the optimal solution and S' be the suboptimal solution that our algorithm produced. Let (i,j) be the first cell where the paths in S and S' diverge. Since S is optimal, the gold collected in S from (i,j) to the end must be greater than or equal to the gold collected in S' from (i,j) to the end. But our algorithm always selects the cell with the maximum gold from the three possible moves, so it follows that the gold collected in S' must be greater than or equal to the gold collected in S from (i,j) to the end. This contradicts our assumption that S is optimal, and thus our algorithm must produce the optimal solution.

The complexity of our algorithm is O(nm), where n is the number of rows and m is the number of columns in the field. This is because we visit each cell exactly once and perform constant time operations at each cell.Therefore, the time complexity is proportional to the size of the input.