Problem 1. (50 points)

Idea (10 points): In order to find a subset of edges $S$ such that every cycle in $G$ contains at least one edge in $S$ and we want the total weight of $S$ to be as small as possible, we can find a maximum spanning tree of graph $G$. The difference between the edges in $G$ and the edges in the maximum spanning tree is our subset of edges $S$.

1. We want to negate all edge weights and then apply the MST algorithm rule. That is, multiply the negative value (-1) to all edge weights.
2. Then apply either Kruskal’s or Prim’s algorithm to find the minimum spanning tree.
3. The result of minimum spanning tree is the maximum spanning tree of the graph.
4. Calculate $S$ based on the difference between edges of $G$ and edges of maximum spanning tree.

Algorithm (25 points):

```python
def maxSpanningTree(vertex, edges):
    # loop through edges; set wt to be negative
    for edge in edges:
        edge.weight = (-1) * edge.weight
    # calculate the minimum spanning tree by calling either Kruskal or Prim and store
    # the result into "mst" variable.
    mst = Kruskals(vertex, edges)
    # store all the edges to max variable
    max = edges
    # result = mst
    for edge in mst.edges:
        max.remove(edge)  # remove the min edge from max
    return max  # which is S here.
```

Proof (10 points):

1. Here, negating all the edge weights and running the Kruskal’s algorithm will produce the maximum spanning tree since it ensures the largest edge weights.
2. We can ensure the total weight of $S$ is as small as possible since we construct the maximum spanning tree of the original graph $G$.

Time Complexity (5 points):

1. Negating all the edge weights takes $O(E)$ time
2. Use of Kruskal’s algorithm takes $O(E \log V)$.
3. Determine all edge weights that is not present in the minimum spanning tree takes $O(E)$ time.

Therefore the overall running time for this problem is $O(E\log V)$. 

Problem 2. (50 points)

**Idea (15 points):**

We will use dynamic programming in this problem.

**Subproblems Definition:**

$T[i]$: maximum profit to be gained by opening some subset of the first $i$ locations. We also store $R[i]$ which is 1 if there is a restaurant at location $i$ and 0 otherwise.

**Recursive Formulation:**

Base case: if $i = 0$, then there’s no location available to choose to open a restaurant. $T[0] = 0$.

General case: if $i > 0$, then we have two options. They are

1. **Do not open a restaurant at location** $i$: If we choose to not open a restaurant at location $i$, then the optimal value will come about by considering how to obtain total profits from the best legitimate configuration using the remaining location up to $i-1$. This is just $T[i-1]$.

2. **Open a restaurant at location** $i$: If we open a restaurant at location $i$, then we gain the expected profit $p_i$. As we want to build a restaurant at location $i$, then the closest location to build another restaurant should be at $c_i$, where $c_i$ denote the maximum $j$ which $m_j \leq m_i - k$. This is just $p_i + T[c_i]$.

Since these are the only two possibilities, we can see that we have the following rule for constructing $T$: $T[i] = \max\{ T[i-1], p_i + T[c_i] \}$ if $i > 0$.

If $T[i] = T[i-1]$, then $R[i] = 0$ and $R[i] = 1$ otherwise.

Note we can compute $c_i = \max \{ j : m_j \leq m_i - k \}$ for every $i$. Note that for some values of $i$ and $c_i$ may not exist in which case, we assume that $c_i = 0$. 

Algorithm (30 points):

```python
# Algorithm to compute c_i for every i
def Compute_ci(ml, ..., mn, k):
    initialize m_i' = m_i - k for all i
    Merge the sorted arrays {m_1, ..., m_n} and {m_1', ..., m_n'} into A
    (when there is a tie in comparing m_i and m_j' for some i and j, put
    m_j' into A first)
    t = 0
    for j = 1 to 2n:
        if A[j] == m_i' for some i
            set c_i = t
        else:
            t++

# Algorithm to find optimal profit and locations to open restaurants
def Find_Optimal_Profit_And_Pos(ml...mn, pl...pn, c_1...c_n):
    T[0] = 0
    for i = 1 to n:
        Not_Open_At_I = T[i-1]
        Open_At_I = p_i + T[c_i]
        if Not_Open_At_I > Open_At_I:
            T[i] = Not_Open_At_I
            R[i] = 0
        else:
            T[i] = Open_At_I
            R[i] = 1
    return T[n] and R

# Algorithm to report optimal locations to open restaurants
def Report_Optimal_Locations(R, c_1...c_n):
    j = n
    S = empty set
    while j >= 1:
        if R[j] = 1:
            Insert m_j into S
            j = c_j
        else:
            j = j - 1
    return S
```

Time Complexity (5 points):

The Compute_ci takes O(n) time to compute c_i for every i. (binary search will increase this to O(log n)).

The Find_Optimal_Profit_And_Pos takes O(n) time to compute T and R.

The Report_Optimal_Locations(R, c_1...c_n) takes O(n) time to report the optimal locations for opening restaurants.

Therefore, the overall running time is O(n).