Part 1)
Code :

```
def optimalPath (hotelList) :
    for i in range(1,len(hotelList)) :
        min = optPenalty[0]+(200-(hotelList[i]-hotelList[0]))**2
        index =0
        for j in range(1,i) :
            value =  optPenalty[j]+(200-(hotelList[i]-hotelList[j]))**2
            if value < min :
                min = value
                index =j
        optPenalty.append(min)
        optPath[i] =optPath[index].copy()
        optPath[i].append(i)
```

optimalPath function takes a list that contains hotel distances from start point to the hotels and returns the optimum path and its total penalty for the last hotel. For n. hotel, the optimum penalty is found by looking at from 0 to (n-1). hotel's optimum penalty and plus n. hotel distance penalty.

$$\text{OptPenalty(n)} = \min_{0<i<(n-1)} \{OptPenalty(i)+(200-(hotelDistance[n]-hotelDistance[i]))^2\}$$

We look from 0 to (n-1). hotel minimum penalty and add n. hotel penalty to it. Path for minimum penalty is the optimum path for n. hotel.

So for i. hotel we have to look (i-1) hotel

$$\sum_{i=0}^{n}\sum_{j=0}^{i-1} 1 = \sum_{i=0}^{n}(i-1) = \frac{n*(n-1)}{2} = O(n^2)$$

Part 2)
Code :

```
def splitValidWords (dict,text) :
    splitPoint =[False]*(len(text)+1)
    words =[]
    splitPoint[0] =True
    for i in range(0,len(text)+1):
        for j in range(0,i) :
            if splitPoint[j] and check(dict,text[j:i]) :
                splitPoint[i] =True
                words.append(text[j:i])
```

splitValidWords function takes a dictionary and a corrupted text and split valid words inside the corrupted text and returns the valid words. To split valid words from corrupted text at index i, we need to find a valid split point at index j, j < i , so that splitPoint[j] is true and text[j+1:i] is valid. For this, splitPoint[0] is assigned true. When a valid words is encountered, splitPoint at that index is assigned true. For index i, look from 0 to (i-1) to find the valid word. But first checks that If there is a valid split point at that index. So time complexity for this algorithm is :

$$\sum_{i=0}^{n}\sum_{j=0}^{i-1} 1 = \sum_{i=0}^{n}(i-1) = \frac{n*(n-1)}{2} = O(n^2)$$

Part 3)
Code :

```
def merge(arr1, arr2) :
    output =[]
    if len(arr1) ==0 :
        output.extend(arr2)
        return output
    if len(arr2) ==0 :
        output.extend(arr1)
        return output
    if arr1[0] <arr2[0] :
        output.append(arr1[0])
        arr1.pop(0)
    else:
        output.append(arr2[0])
        arr2.pop(0)
    output.extend(merge(arr1,arr2))
    return output

def combineArrays(list) :
    if len(list) == 1 :
        return list.pop()
    result =merge(list.pop(),list.pop())
    list.append(result)
    return combineArrays(list)
```

combineArrays function takes a list of arrays and combine them into a single sorted array. Both combineArrays and merge function use the divide and conquer technique. CombineArrays functions takes 2 arrays from the list and send them to merge function. Merge function takes 2 arrays and merge them in a sorted way. After merging 2 arrays combineArrays function takes the result into the list and call itself with that list. CombineArrays function repeats this until 1 array left in the list and return that array.
merge function sorts 2 arrays by comparing the first items of both arrays and puts the smaller one into the output list and remove that item from the array and then call itself with that new arrays. Merge function repeats this until one of the arrays becomes empty and then appends other list into the output list and returns output list.

Time complexity for merge function :

$T(n,n) = T(n-1,n-1) +1$ $\quad T(1,1) =1$

$=T(n-2,n-2) +1 +1$

$=T(n-3,n-3)+1+1+1$

.

.

$= T(n-p,n-p) +p*1$ for $P =n-1$

$=T(1,1)+ (n-1)*1$

$= 1+n-1 = O(n)$

Time complexity for combine function :

$T(k) =T(k-1)+O(2n)$

$= T(k-2) +O(3n) + O(2n)$

$=T(k-3)+ O(4n) + O(3n) +O(2n)$

.

.

$=T(k-p) + O((p+1)*n)+O(p*n)+O((p-1)*n) \ldots +O(2n)$ for $p=k-1$

$=T(1) + O(kn)+O((k-1)*n)+O((k-2)*)\ldots.. +O(2n)$

$= O(kn)$

Part 4)
Code :

```
def inviteList (graph) :
    graph =graph.copy()
    people =list(graph.keys())
    size =len(graph.keys())
    for i in people :
        if len(graph[i]) <5 or size-5 < len(graph[i]) :
            graph.pop(i)
            for j in graph.keys() :
                if i in graph[j]:
                    graph[j].remove(i)
    return list (graph.keys())
```

inviteList takes a graph that shows the people and their connection. If a person knows someone then they are connected each other in the graph in both direction. For invite list we have to choose people that have at least 5 connection which means knows at least 5 people and at most total people -5 (size-5) connection which means that person does not know at least 5 people that are invited. Size variable in inviteList function is the total number of people and len(graph[i]) will give us the number of people that i. person knows.
We will remove the person that have connection smaller than 5 or bigger than size-5. So that remaining people in the graph can be invited to the party. After removing a person from graph, we are removing that person from other people's list too. So after removing, we have to check other members of the graph.

Time complexity for this algorithm :
For n input we have to check each one which is O(n) but if there is a person that we have to remove, we have to revisit all graph element to remove that item which increases the time complexity to O(n^2)

Part 5)
Code :

```
def isConnected (graph,x,y) :
    queue = deque()
    visited = [False] * (len(graph) + 1)
    queue.append(x)
    visited[x] = True
    while (0 <len(queue)):
        vertex = queue.popleft()
        for s in graph[vertex]:
            if y == s :
                return True
            if visited[s] == False:
                queue.append(s)
                visited[s] = True
    return False
def checkConstraints (equality,inequality) :
    while 0 <(len(inequality)) :
        list =inequality.pop(0)
        x =list.pop()
        y =list.pop()
        if isConnected(equality,x,y) :
            result =[False]
            result.append([y,x])
            return result
    result =[True]
    return result
```

Equality relationship is represented in a graph like that. If there is an equality like that X1 = X2 then they should be connected in the graph but if there is an inequality like that X3 != X4 then they should not be connected in the graph. Inequality relationship is represented as a list like (X3,X4). CheckConstraints function takes an equality graph and inequality list. Constraints are satisfied if there is no connection between the inequality relationship. Function takes an inequality constraint and check if its member is connected in the equality graph. If they are connected then this constraint is not satisfied and function return false. If Every inequality list is satisfied then constraints is satisfied and function return true.

IsConnected function takes a graph and 2 member and checks if they are connected or not. Function uses the breadth-first search algorithm technique. It looks every vertex around the root vertex if it can not find there then move the another vertex.

Time complexity for isConnected :

isConnected function uses the breadth-first search algorithm. For N vertex and M edges which represents equality relationships. If there is a connection between x any y then function returns but in the worst case function have to visit all nodes, which is O(N+M) time.

Time complexity for checkConstraints :

checkConstraints function have to look every member of inequality list to check that if constraints are satisfied or not. If one constraint does not hold then function returns. But for P inequality relationship, N is equality graph vertex and M is equality relationship (edges) then function will run at most O(P*(N+M)) time