1. Algorithm:
   Sort set A and Set B in ascending order. And pair A[i] with B[i] to get the maximum payoff.

```
A.sort()
B.sort()
Pay = 0
For i in range(len(A)):
        Pay *=  A[i] ** B[i]
Return pay
```

   Proof:
   Prove by contradiction: In the algorithm A[i] < A[j] and B[i] < B[j] and payoff = A[i] ** B[i] + A[j] * B[j]. Assume there is a method that can generate a larger payoff. There must be at one pair of numbers that is different from the one in the algorithm such as A[i] < A[j] and B[i] > B[j]
   For A[i] < A[j] and B[i] > B[j] in the optimal method is smaller than the ones in the algorithm:
   (A[i] **B[j] * A[j] **B[i])(optimal) / (A[i] ** B[i] * A[j] ** B[j]) (algorithm) = A[i] ** (B[j] - B[i]) / A[j] **(B[j] - B[i]) < 1 because of A[i] < A[j]
   The optimal method cannot do worse than the algorithm. So by contradiction, the algorithm is optimal.

   Time Complexity is O(nlog(n)). Sorting A and B takes nlogn time, for loop takes linear time. Total time complexity is O(nlogn).

2. D is already sorted by the distance. We need to check if we can get to the next gas station, if we can, continue driving. If we cannot, we need to refill at the current gas station. Repeat the process until we get to the USC.

```
Last_stop = 0
Gas_stop = []
For i in range(len(d)):
        If d[i] - last_stop > p:
                gas_stop.append(d[i-1])
                Last_stop = d[i-1]
Return gas_stop
```

   Proof:
   Proof by contradiction: Assume the optimal method can do better than the algorithm. If the optimal method stop for gas before d[i] (the stop point in the algorithm) then it must have more stops because it cannot reach the next stop as far as the algorithm does. If the optimal method stops after the d[i] (the stop point in the algorithm) it is impossible because it will not be able to reach any stop point after i without refilling before i. So the algorithm is optimal.

Time complexity: O(n)

3. 1. Iterate over S', if we find the first element in S, we start looking for the next element of S in S'. After we check all the elements in S', check if we have had all the elements in S. if we have all elements of S appearing in the same order in S', print Yes.

```
check_ith_event = 0
For event in S':
        If event != S[check_ith_event]:
                continue
        If event == S[check_ith_event]:
                check_ith_event += 1
        If check_ith_event == len(S) - 1:
                print("Yes")
                break
```

   Time complexity is O(m+n) since we iterate lists m and n only once.

   Proof by induction:
   Base case: one event in S. If the event appears in S', S' is a subsequence of S.
   Induction hypothesis: Let us assume that the algorithm has found the r-1th event in S.
   Induction step: Show the algorithm can find the rth event in S
   When the algorithm finds the r-1 event in S, the check_ith_event will increment by 1, when the event in S' is not the rth event in S, the for loop continues. If the event in S' is the same as the rth event in S, we have found the rth event in S in S'. we further check if rth event is the last event in S. if it is the last event, we have found a subsequence of S in S'.

4. Prove its optimality by induction
   Base case: if there is the total weight of all the packages is less than W, we only need one truck.
   IH: assume it is true for K-1 trucks.
   Induction step: prove it is true for K trucks. Assume there is an optimal method that can do better than the greedy algorithm.
   Case 1, it could do better by putting fewer boxes on the K-1 truck. If we put fewer boxes on the K -1 truck, we do not put as many boxes as the greedy algorithm. The greedy algorithm will need fewer trucks.
   Case 2 if it could do better by putting more boxes on the K-1 truck. It is also impossible because the greedy algorithm makes sure that the K-1 truck cannot load any more boxes.
   So the greedy algorithm is the optimal solution.