Problem 1

Algorithm:
Every day suppose start from d_i to d_j when d_j -d_i <K and d_{j+1}- d_i>K. Go as far as possible until the gas is not enough to go to next gas station.
Time complexity:
O(n).
Prove optimal:
Let G= {G_1, ..., G_k} denote the set of stopping points chosen by the greedy algorithm, and suppose by way of contradiction that there is an optimal solution containing less stopping points; let's call this smaller set O= {O_1, ..., O_m}, with $m < k$.
To obtain a contradiction, we first show that the stopping point reached by the greedy algorithm on each day j is farther than the stopping point reached under the alternate solution.
For each j=1, 2..., m, we have $G_j \geq O_j$.
The case j=1 in the greedy algorithm, we drive as far as we can on the first day before stopping. Now let $j > 1$ and assume that the claim is true for all $i < j$.
In the optimal solution, $O_j - O_{j-1} \leq K$. By our assumption, we have $G_{j-1} > O_{j-1}$.
Thus, O_j-G_{j-1}<O_j-O_{j-1} ≤K. It means the distance from the stopping point on day j-1 in greedy algorithm to the stopping point on day j in optimal algorithm is less than K. According to the greedy algorithm, our stopping point can only be farther than O_j.
For day m, we have $O_m \leq G_m$. If m<k, it means G_m < d-K. Otherwise there is no need for another stopping point G_{m+1}. So, we get $O_m \leq d$-K. This contradicts the assumption from the beginning that the optimal solution is a valid solution. It must have more stopping points than m. So, we cannot have m<k.
Now we proved that the greedy algorithm is optimal.

Problem 2

1.
Optimal structure:
For each integer x, the only possible ways to write x using 1,3 and 4 are ways_of_writing(x-1)+1, ways_of_writing(x-3)+3, ways_of_writing(x-4)+4, ways_of_writing(x-4) +1+3, ways_of_writing(x-4) +3+1.
If we want to compute number of ways to write integer x, we must compute the number of ways of integer x-1, x-3 and x-4. Suppose we optimally compute the number of different ways to write x-1, x-3 and x-4, Then we can get No_diff_ways (x) by the sum of 1 * No_diff_ways (x-1), 1 * No_diff_ways (x-3), and 3 * No_diff_ways (x-4).
Recursive expression:
Base case: x=0, F (0) =0;
No_diff_ways (x) = [1 * No_diff_ways (x-1)]
+ [1 * No_diff_ways (x-3)]
+ [3 * No_diff_ways (x-4)]

Bottom-up algorithm:

```
Input: integer $x$
Output: number of different ways to write $x$
Init: No_diff_ways [0] =0
For $i$=0 to $x$ do:
    No_diff_ways [$x$] = [1 * No_diff_ways [$x$-1]]
                       + [1 * No_diff_ways [$x$-3]]
                       + [3 * No_diff_ways ($x$-4)]
End
Return No_diff_ways [$x$]
```

Time complexity:

O(x). Each subproblem corresponds to an integer from 1 to x, of which there are x subproblems. In each subproblem, we find 3 values from the array No_diff_ways[], which takes time O(1). This results in time complexity O(1*x) =O(x).

Space complexity:

O(x). The array No_diff_ways exactly stores x values.

2.

Optimal structure:

Each time Alan has two options, to take the first or last gift to gain value v(*first*) or v(*last*). After Alan's option, Charlie also has two options to take the first or last gift from remaining gifts. There are totally 4 possible values when Alan picks up a gift from $v(i)$ to $v(j)$:

When Alan picks the first one and Charlie picks the last one from remaining gifts:

$v(i)$ + Max_total_value (i+1, j-1),

When Alan picks the first one and Charlie picks the first one from remaining gifts:

$v(i)$ + Max_total_value (i+2, j),

When Alan picks the last one and Charlie picks the last one from remaining gifts:

$v(j)$ + Max_total_value (i+1, j-1),

When Alan picks the last one and Charlie picks the last one from remaining gifts:

$v(j)$ + Max_total_value (i, j-2)

Suppose we have the optimal maximum possible total value for Max_total_value (i+1, j-1), Max_total_value (i+2, j), Max_total_value (i, j-2), then we can choose the largest value from the above 4 possible values to get the maximum total value.

If there is a value, *NewLargerValue*, larger than the maximum total value we just computed, we can substitute

Max_total_value (i+1, j-1) by *NewLargerValue* – max[$v(i)$, $v(j)$],

Max_total_value (i+2, j) by *NewLargerValue* – $v(i)$,

Max_total_value (i, j-2) by *NewLargerValue* – $v(j)$

to set larger values for the gift lists $v(i+1)$ to $v(j-1)$, $v(i+2)$ to $v(j)$ and $v(i)$ to $v(j-2)$which contradicts our assumption that the values of the 3 lists are already maximum.

Recursive expression:

Base case: n=2. Max_total_value(n) = max [$v(1)$, $v(2)$]

Max_total_value (i, j) =
max [$v(i)$ + Max_total_value $(i+1, j-1)$, $v(i)$ + Max_total_value $(i+2, j)$,
$v(j)$ + Max_total_value $(i+1, j-1)$, $v(j)$ + Max_total_value $(i, j-2)$]
where i, j =0, 1…n.

Bottom-up algorithm:

```
Input: a list of value $v(1), \ldots, v(n)$
Output: the max sum of value Max_total_value [1, $n$]
Max_total_value [1,2] = max [$v(1)$, $v(2)$]
For $i$ from 1 to $n$-1 do:
    For ($j = i+1; j \le n; j+2$) do:
        Max_total_value [$i, j$] =
            max [ $v(i)$ + Max_total_value ($i+1, j-1$), $v(i)$ + Max_total_value ($i+2, j$),
            $v(j)$ + Max_total_value ($i+1, j-1$), $v(j)$ + Max_total_value ($i, j-2$)]
End
Return Max_total_value [$i, j$]
```

Time complexity:

O(n^2). For an input integer n, there are $n*(n/2) = n^2/2$ loops. In each loop the algorithm looks up for 4 values in the array Max_total_value, which takes time O (1).

Space complexity:

O(n^2). For each integer n, Max_total_value [1, n] exactly stores $1+3+5+\ldots+(n-1) = n^2/4$ values.

Problem 3

1.

(a)

1. Add 1 capacity on the edge (u,v) in the residual graph of G on the given maximum flow.
2. Using BFS to search an augmentation path.
   In BFS, setting a vertex or edge label takes O(1) time. Each vertex is labeled twice, once as unexplored and once as visited. Each edge is labeled twice, once as unexplored and once as discovery or cross.
   Time complexity of BFS is $O(|V|+|E|)$.
3. If an augmentation path exists, increase the maximum flow by 1.
   If no augmentation path exists, the maximum flow keeps unchanged.

(b)

1. Draw the residual graph of G on the given maximum flow.
2. If the edge (u,v) has positive residual capacity, decrease the residual capacity by 1. Maximum flow keeps unchanged.
3. If the edge (u,v) has negative residual capacity, increase the residual capacity of its reverse edge by 1, use BFS to search an augmentation path going from t to s which includes the decreased edge. Time complexity is $O(|V|+|E|)$. If an augmentation path exists, decrease the maximum flow by 1. If no augmentation path exists, the

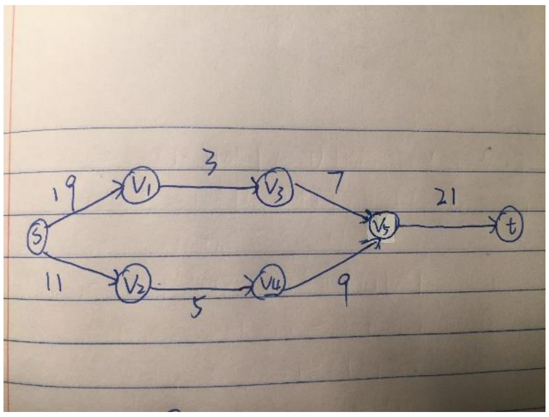
maximum flow keeps unchanged.

Problem 4

1. *If the capacity of every arc is even, then the value of the maximum flow must be even.*
True. From max-flow min-cut theorem, the value of the max flow is equal to the value of the min s-t cut. The capacity of a cut (A,B), denoted as $c(A,B)$, is the sum of the capacities of all edges out of A. If the capacity of every edge is even, the sum of any sets of edges will be even. Thus, the minimum cut capacity is even. Maximum flow is even.

2. *If the capacity of every arc is even, then there is a maximum flow in which the flow on each arc is even.*
This is true. The maxflow itself is even: the capacity of any cut, including that of a minimum cut, is a sum of even numbers and thus even. Now consider the running of Ford-Fulkerson algorithm. First we choose a path and push the minimal edge capability in this path, which is even. Then we find residual paths. Every Push will push the capacity of an edge (an even number), or a residual capacity (the difference between a capacity or residual capacity, also even). Hence we always push even values of flow; as the initial flow values are even on the edges, the flow is even along every edge at each step including the final one, which yields a flow.

3. *If the capacity of every arc is odd, then the value of the maximum flow must be odd.*
False. Counterexample as the picture above. Maximum flow in this graph is as the minimum cut capacity, which is 8.



4. *If the capacity of every arc is odd, then there is a maximum flow in which the flow on each arc is odd.*
False. Counterexample as the picture above. On the edge (V_5, t) the flow is 8, an even integer.

Problem 5

1. *Locate all the sources and targets in C.*
Use BFS in the graph. For each vertex, look at all its edges. If a vertex only has incoming edges, set this vertex to sources. If a vertex only has outgoing edges, set this vertex to targets. This algorithm has to look at all vertexes and edges in the graph twice.

So the time complexity is O(|V|+|E|)

2. *Prove that a DAG must contain at least one source and at least one target.*
Suppose the DAG has no source. It means every vertex in this graph has at least one incoming edge. For each vertex, move backwards through its incoming edge. There are only *n* vertices overall. If walk backwards *n* times, we must hit a vertex we have seen before. Thus, there is a cycle and the graph is not a DAG. Same as the target node.

3. *Let $G = (V,E)$ be a DAG. Propose an $O(|V|+|E|)$ time algorithm to count the total number of paths from all the sources in G to all the targets in G.*
Input: a DAG *G*, a list of sources *S*, a list of targets *T*
Output: number of paths from all sources in *G* to all the targets in *G*

```
Num_of_path = 0
Num_of_path[][]=0
For a vertex $s_i$ in $S$ do:
|       For every vertex $t_i$ in $T$ do:
|           Num_of_path += Number_of_Path($s_i$, $t_i$)
End

Number_of_Path($u$, $v$):
|   For all vertices $w_i$ whose incoming edge is an outgoing edge of $u$ do:
|       Num_of_path[$u$][$v$] += Number_of_Path($w_i$, $v$)
Return Num_of_path[$u$][$v$]
```

Time complexity:
We record in the array Num_of_path[][] the number of paths from each non-target vertex to each target. We visit each vertex and each edge in the graph exactly once. Thus, time complexity is $O(|V|+|E|)$.