Q1:

(a) $3^{\log_{10}(n)} < \log_{10}(\pi^n) < n \cdot \log_{10}(n) < n^{2017} < n^{\log_{10}(n)} < 1.01^n$

(b) $f(n) = \sum_{i=0}^{\lfloor \log_3^n \rfloor} 3^i \leq 1 + 3^1 + \cdots + 3^{\log_3^n} = 1+3+9+\cdots+n \in O(n) \in O(0.5n)$

$\therefore f(x) \in O(g(x))$

(c) $\lim_{n\to\infty} \frac{f(x)}{g(x)} = \lim_{n\to\infty} \frac{4^{n^2}}{100^n} = \lim_{n\to\infty} \frac{\log 4^{n^2}}{\log 100^n} = \lim_{n\to\infty} \frac{n^2 \cdot \log 4}{n \cdot \log 100} = \infty$

Thus, $f(x) \in \omega(g(x))$

(d) No. Counter example:

$f(x) = 2n\log n + n \in \Theta(n\log n + n)$

$g(x) = n\log n \in \Theta(n\log n)$

$f(x) - g(x) = n\log n + n \in \Theta(n\log n) \in \omega(n)$

Q2:

Assume for $M_1, M_2, \ldots, M_n \in$ list1, $rank(M_1) > rank(M_2) > \ldots > rank(M_n)$

Assume the original sequence of list 2 is the same as list1. Now, to count the conflicts is equivalent to count the steps needed for sorting list2 to rank-decreasing order.

To meet the time complexity requirement, we will consider merge-sort as the sorting algorithm

Thus, we have the following algorithm based on merge-sort.

```
mergeCountConflicts(list):
    n ← list.size()
    l1 ← list[0:n-1]
    l2 ← list[n:]
    count ← mergeCountConflicts(l1) + mergeCountConflicts(l2)
    i ← 0
    j ← 0
    list.clear()
    while i < l1.size() and j < l2.size() do:
        if i == l1.size() then
            list.append(l2)
            break
        else if j == l2.size() then
            list.append(l1)
            break
        else if rank(l1[i]) > rank(l2[j]) then
            list.push(l1[i])
            i++
        else
            list.push(l2[j])
            j++
            count += l1.size()
        end if
    end while
    return count

DC-countConflicts(list1, list2):
    return MergeCountConflicts(list2)
```

Analysis:

Assume in list 2, $rank(M_i) = j$, then M_i will be placed as jth movie in the sorted list, and along the process of moving M_i to $M_{(j)}$, it will meet $j-i$ # movies which means it has conflict with each one of them.

In mergeCountConflicts, we expect l2 is smaller than l1, so every time we push an element into list, we actually move that element l1.size() # forward. Thus, count += l1.size().

Complexity analysis:

$T(n) = P(n)$ and $P(n) = 2 \cdot P(\frac{n}{2}) + O(n)$ while loop total n times.

By master theorem, $T(n) \in O(n \cdot \log(n))$.

Q3:

Assume the input set of intervals is I, and for each input interval i, $i = [begin(i), end(i)]$.
Then we design the algorithm as follow:

```
greedyCheck( I ):
    cur <- begin( I[0] )
    for i = 1 to n-1
        if cur + 2 ≤ begin( I[i] )
            cur = begin( I[i] )
        else if cur + 2 ≤ end( I[i] )
            cur = cur + 2
        else
            return NO.
    end loop
    return YES
```

Complexity analysis:

$T(n) = O(n)$, since we only iterate n times at worst case to check each interval in input I.

Suppose S_g is the greedy solution set of points and S is some other hypothetical optimal solution

Claim:

For every k-th point in S_g, it is less than k-th point in S.

Proof of Optimal:

By our claim, we know that whenever we want to pick the $(k+1)$th point, S_g will have a wider range to pick point. This means if S_g does not exist, S does not exist neither. Thus, S_g is the optimal solution.

Proof of Claim:

- Base case:

  $i=0$, $S_g[0] = begin(I[0]) \le S[0] \le end(I[0])$. Holds

- IH:

  $i=k$, $S_g[k] \le S[k]$

- IC:

  $i=k+1$, there are two cases:

  ① $S_g[k] + 2 \le begin(I[k+1])$

  $S_g[k+1] = begin(I[k+1]) \le S[k+1] \le end(I[k+1])$

  ② $begin(I[k+1]) \le S_g[k]+2 \le end(I[k+1])$

  $S_g[k+1] = \underbrace{S_g[k]+2 \le S[k]+2}_{IH} \le S[k+1]$

  Thus, IH holds.

Therefore, by POMI, we prove that $S_g[i] \le S[i]$ for $i=0,\dots,n-1$.

Q4:

Let $A = [a_1, \cdots, a_n]$.

We will design the dynamic-programming algorithm as follow:

```
minimize (A, i, k, W, set, map, push):
    if map(i, k) does not exist then:
        if (push) then
            set.push(A[i-1])
        if i = A.size or k=0 then
            return abs(W)
        s1 ← set
        s2 ← set
        a ← find(A, i+1, k, W, s1, 0)
        b ← find(A, i+1, k-1, W-A[i], set2, 1)
        if a ≤ b then
            set ← s1
            map(i,k) = a
        else
            set ← s2
            map(i,k) = b
    return map(i,k)
DP-minimize (A, k, W):
    // set : vector of numbers, solution set.
    // map: hash table to store map(i,k) = minimize(A, i, k, W, set, map)
    minimize (A, 0, k, W, set, map).
    return set.
```

Correctness analysis:

A set $A = \{a_1, \ldots, a_n\}$, for each a_i, only 2 possibilities:

① $a_i \in S$. then $k-1$ and $W - a_i$

② $a_i \notin S$. then k and W

we will compare the result of these two to consider wether include a_i or not.

Complexity analysis:

$T(n) = P(n)$ and $P(n)$ is for minimize$(\cdots)$.

Since we use hash table to store each result of (i,k), whenever map(i,k) exists, we only need $O(1)$ to read that data. Therefore, in total, we need at least $n \cdot k$ times to calculate all levels of results. Then, $P(n) \in O(n \cdot k)$. Since k and W are positive integer, from the question we get that $k \le W$. Therefore, $T(n) \in O(n \cdot W)$.

Q5:

We will design the following DFS algorithm:

```
DFS(G, v, path):
    mark v as visited
    for u in v.neighbor() do
        if u is not visited. then
            mark u as visited
            path.push(u)
            DFS(G, u, path)
            path.pop()
        else
            mark (path[path.size()-1], u) as circular
            for p = path.size()-2 down to 0 do
                mark (path[p], path[p+1]) as circular
                if path[p] = u then
                    break
            end loop
        end if
    end loop.
CP(G):
    // V: the set of vertices, E: the set of edges.
    // path: empty set of edges.
    for v in V do
        if v is not visited then
            path.clear()
            DFS(G, v, path)
    end loop
    return all edges marked as circular.
```

Analysis:

We know that DFS algorithm checks the sub-vertex of every path, thus such path that forms circular will be found. The time complexity of DFS is $O(m+n)$, and for marking the edges as circular, it will require k times to call and $(k-1)$ times for every edge connecting it. Thus, $T(n) \in O(m+n)$