Q1:

i)

We have a polynomial time verifier: Since CLIQUE and IS are in NP, they both have a polynomial time verifier. If both verifiers return true, the algorithm finds a clique with size at least k and an independent set with size at least k, CliqueAndIS is true.

```
Checker-CliqueAndIS(G,k){
      bool C = Checker-CLIQUE(G,k);
      bool I = Checker-IS(G,k);
      return (C && I);
}
```

Checking takes $\text{polynomial} + \text{polynomial} = \text{polynomial}$ time. Therefore, CliqueAndIS is in NP.

Clique algorithm can polynomial reduce to CliqueAndIS algorithm:

```
Clique(G,k){
      bool answer = CliqueAndIS(G,k);
      return answer;
}
```

Since CLIQUE reduces to CliqueAndIS, CliqueAndIS is in NP-complete.

ii)

We have a polynomial time verifier: for every $u, v \in V'$, $(\pi(u), \pi(v)) \in E$ if and only if $u, v \in E'$.

```
Check-Subgraph(G,H){
      for (u = 1; u < |V'|; ++u){
            for (v = 1; v < |V'|; ++v){
                  if (u == v) {
                        continue;
                  } else if ((p(u), p(v)) ∈ E) && ((u, v) ∈ E'){
                        continue;
                  } else if ((p(u), p(v)) ∉ E) && ((u, v) ∉ E'){
                        continue;
                  } else {
                        return false;
                  }
            }
      }
      return true;
}
```

Checking takes polynomial time. Therefore, Subgraph is in NP.

If we let H be a clique of size k, then Clique algorithm can polynomial reduce to Subgraph:

```
Clique(G,k){
      H = build_clique(k);
      bool answer = Subgraph(G,H);
      return answer;
}
```
Therefore Clique reduces to Subgraph, Subgraph is in NP-cpmplete.

Q2:

a)

We have a polynomial time verifier, which loops over all clubs check if there is a person form that club.

```
Checker-Club(n,C,k){
      group[k] = set of people that satisfies the requirement.
      for each club in C{
            for (int person = 1; person <= k; ++person){
                  if (group[person] ∈ club){
                        goto NEXT_CLUB;
                  } else {
                        continue;
                  }
            }
            // one club did not contain person in possible answer
            // checking did not pass.
            return false;
NEXT_CLUB:
      }
      // every club contains at least one person in possible answer
      // answer is true.
      return true;
}
```

There are $|C|$ clubs, for each club we need to check at most k people, each check takes at most time n ,which is the maximum size of any club.

Checking takes : $n \times k \times |C| \in Polynomial$ time. Therefore, Club is in NP.

VertexCover algorithm can polynomial reduce to Club algorithm:
Assume we are solving VertexCover(G,k). For each vertice in G, we build a club that has itself(call it "center") and its adjacent vertices. For a set of vertices called S, if a club contains a vertex in S, then club's center is covered by that vertex in S. If we ask whether every club contains at least one of the vertex in S, and all of them say true, then every center is covered by a vertex in S. Since every vertex in G produces a unique club, and that club's center is covered, we know every vertex is covered by S, S is a vertex cover of graph G.

```
VertexCover(G,k){
      clubs C[|V|];
      for (int i = 1; i <= |V|; ++i){
            C[i] = {i, adjacent vertices of i};
      }
      return Clubs(n,C,k);
}
```

Since VertexCover reduces to Club, Club is in NP-complete.

b)

We use a helper function SUM(S), which takes an array S and returns the sum of all elements in S.

We have a polynomial time verifier, which separately sums two sets returned by EvenSplit and check if they are equal.

```
Checker-EvenSplit(A,B){
      int a = SUM(A);
      int b = SUM(B);
      return (a == b);
}
```

Checking takes : $|A| + |B| = |S| \in Polynomial$ time. Therefore, EvenSplit is in NP.

SubsetSum algorithm can polynomial reduce to EvenSplit algorithm:
To calculate SubsetSum(S, k), first take the sum of S and call it SUM. let $S' = S \cup \{2k - SUM\}$, We know the elements in S' sums to $SUM + 2k - SUM = 2k$. If we can partition S', we have two sets each sums to k. Since only one set contains $2k - SUM$, the other set which is the subset of S sums to k, and SubsetSum(S, k) returns true.

```
SubsetSum(S,k){
      int n = |S| + 1;
      int S'[n];
      for (int i = 1; i <= n - 1; ++i){
            S'[i] = S[i];
      }
      S'[n] = 2k - SUM(S);
      return EvenSplit(S');
}
```

Since SubsetSum reduces to EvenSplit, EvenSplit is in NP-complete.

Q3:

a)

We have a polynomial time verifier: Assume CLIQUE3 finds clique $C = (V', E')$. for every $u, v \in V'$, if $u, v \in E$, then C is a clique of at least size k.

```
Checker-CLIQUE3(C){
      for (u = 1; u < |V'|; ++u){
            for (v = 1; v < |V'|; ++v){
                  if (u == v) {
                        continue;
                  } else if ((u,v) ∈ E) {
                        continue;
                  } else {
                        return false;
                  }
            }
      }
      return true;
}
```

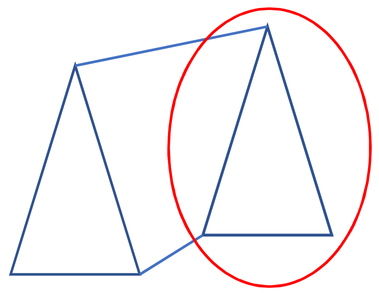
Checking takes : $|\mathrm{V}'| \times |\mathrm{V}'| = k^2 \in Polynomial$ time. Therefore, CLIQUE3 is in NP.

b)

The thing we need to show is a polynomial-time reduction from Clique to Clique3. The claim did the inverse.

c)

Consider the following graph, with blue vertices and blue edges:



Let C be set of vertices in the red eclipse. The vertices $V \backslash C$ is a clique in G, but C is not a vertex cover in G.

d)
Here is a polynomial time sudo algorithm that solves CLIQUE3.

```
Clique3(G,k){
      if(k == 4){
            // since all vertices have at most degree 3, there is one and
            // only one graph that has a 4-clique as its subgraph, which is
            // 4-clique itself.
            for (u = 1; u <= 4; ++u){
                  for (v = 1; v <= 4; ++v){
                        if (u == v) {
                              continue;
                        } else if ((u,v) ∈ E) {
                              continue;
                        } else {
                              return false;
                        }
                  }
            }
            return true;
      } else if (k == 2){
            // return true if there is an edge.
            return (|E| >= 1);
      } else if (k == 1){
            // return true if there is a vertex.
            return (|V| >= 1);
      } else if (k >= 5){
            //impossible cases.
            return false;
      } else {
            // k = 3, we need to find a triangle as its subgraph.
            for each edge(u, v){
                  for each vertex w{
                        if ((u,w) ∈ E) && ((v,w) ∈ E){
                              return true;
                        }
                  }
            }
            return false;
      }
}
```

Algorithm takes at most $|E| \times |V| \times \log|E| \in Polynomial$ time, so $CLIQUE3 \in P$.

Q5:
We make modifications over FloydWarshall algorithm. We simultaneously build two tables: one is the original FloydWarshall table, the other is the ConstrainedAPSP table. When updating path in ConstrainedAPSP table, under the condition that we have passed a vertex in S, we try to maximize the use the FloydWarshall path to minimize the ConstrainedAPSP path weight.

```
ConstrainedAPSP (G=(V,E), w, S){
      dist[|V|][|V|][2];
      // stores path length from u to v with mode 1 or 2. Mode 1 is the
      // original FloydWarshall algorithm, mode 2 goes through at least one
      vertex in S.
      //
      for each vertex u in V{
            for each vertex v in V {
                  if ((u,v) ∈ E){
                        dist[u][v][1] = w(u,v);
                        if (u ∈ S || v ∈ S){
                              dist[u][v][2] = w(u,v);
                        } else {
                              dist[u][v][2] = ∞;
                        }
                  } else if (u == v){
                        dist[u][v][1] = 0;
                        dist[u][v][2] = 0;
                  } else {
                        dist[u][v][1] = ∞;
                        dist[u][v][2] = ∞;
                  }
            }
      }
      temp = dist;
      for (int i = 1; i <= n; ++i){
            for each vertex u in V{
                  for each vertex v in V {
                        dist[u][v][1] = min(temp[u][i][1] + temp[i][v][1],
                                            temp[u][v][1]);
                        if(i ∈ S){
                              dist[u][v][2]= min(temp[u][i][1]+temp[i][v][1],
                                                 temp[u][v][2]);
                        } else {
                              dist[u][v][2]= min(temp[u][i][1]+temp[i][v][2],
                                                 temp[u][i][2]+temp[i][v][1],
                                                 temp[u][v][2]);
                        }

                  }
            }
            temp = dist;
      }
      path[u][v] = dist[u][v][2];
      return path;
}
```

The runtime is the same as FloydWarshall algorithm, regardless the choice of set S, which is $\Theta(|V|^3)$.

FloydWarshall path always has shorter weight or equal weight comparing to ConstrainedAPSP path since the latter has contraints.

In ConstrainedAPSP path, we have the invariant that the path recorded always goes though at least one vertex in S.
In initialization, we forced that at least one vertex is in S.
When updating ConstrainedAPSP path, if new vertex c is in S, we will choose FloydWarshall path a->c->b since c is in S and thus it is a ConstrainedAPSP path, or the old ConstrainedAPSP path whichever has less weight. if new vertex c is not in S, we will choose a->c->b with half part ConstrainedAPSP path, so that the whole path is a ConstrainedAPSP path; or the old ConstrainedAPSP path whichever has less weight.
Hence the ConstrainedAPSP table always contains shortest ConstrainedAPSP path.