1. True or False
   1) True P stands for polynomial time solvable problems, and NP contains not only P problems but more harder problems, that is nondeterministic polynomial time solvable problems.
   2) Unknown P stands for polynomial time solvable problems, and NPC stands for the most difficult problems in the domain of NP questions. If P = NP is proved in the future, then NPC ∈ NP = P, otherwise, NPC ∈ NP, P ∈ NP, but there is no gap between NPC problems and P problems.
   3) True SAT ∈ NPC, if there is a polynomial-time algorithm for SAT, then there is a polynomial-time algorithm for every problem in NP
   4) True for all the Boolean variables, there is 1 or 0 for them, so the time complexity is proportional to 2^n, not polynomial.

2. MST with edge {1,2}

Solution:

Alternatively use Prim's algorithm. No need to keep track of components. Prim considers each edge once (assuming adjacency-lists). The most costly part of Prim is looking for the next vertex to add: the one of the remaining vertices that has the cheapest connection to the tree already constructed.

Here we only have to know whether there are still vertices that have a connection of weight 1 (because they have priority to those of weight 2). I would say that a single list can handle this. Thus, the time complexity is O(|V|+|E|)

Another Solution:

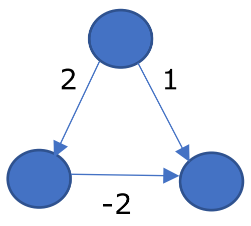
I also think that there might be another approach. Start building spanning trees for the components when we only use edges of weight 1, just ignoring edges of weight 2. Then the components are "big vertices" that can be connected by a tree using the remaining edges of weight 2. Keeping components here is easy: all vertices reached in a new component are labelled with the same number. That number identifies the "big vertex".

3. SSSP

Dijkstra SSSP Algorithm is based on greedy strategy, it finds a nearest point in every searching process, then set this distance as the shortest pace between the nearest point and the source point.

However, if negative-edge exists, there is a possible route that passes through a point which is not the nearest point to the source point, then set a even smaller pace from the source point to the destination point through this negative-edge.

For example, Graph is illustrated below:



The upper vertex is named point A, and for the lower line, the left one is named point B and the right one is named point C.

First, A is set as the source point, then the algorithm will find a point which has the smallest distance to the already processed point set S, so point C is selected by Dijkstra SSSP algorithm, and it is put into the set S with a shortest distance '1' to the source point A, and the shortest pace between A and C will no longer be updated. However, it can be seen that A -> B -> C is a even shorter pace from A to C with the distance of 0.

So, Dijkstra SSSP Algorithm fails in the condition that there is some negative-edge in the Graph.

4. Cycle Detection

Algorithm 01:

Given a cycle of points <1-2-3-4-1> in a Graph G, these points have at least degree-of-2. Based on this, an algorithm for determine whether there is a cycle in a Graph G is introduced (similar to the topological sort in DAG)

Step 1: Get the degree of all the points in Graph G.

Step 2: Delete all the vertices and edges incident on them which degree is 1 or 0. Then, subtract the degree of another vertex incident on the edge by 1.

Step 3: iterate all the vertices, if there has vertex which degree is less than 2, back to step 2.

Step 4: If there is at least one vertex left in the Graph G, output 'CYCLE IN GRAPH', otherwise output 'No CYCLE IN GRAPH'

Time Complexity: O(V+E)

This algorithm can detect whether there is a cycle or not in a given graph G, however, it can't precisely show a cycle in G if there are more than one cycle exists in the graph.

For the aim of show a cycle in the graph G, turn to algorithm 02.

Algorithm 02:

Run DFS Algorithm on Graph G, as we know there are three types of color when doing searching, that is white, gray and black. In the beginning, all the vertices are white, when a vertex is being visit, it will become gray, after all the adjacent vertices of the gray vertex are visited, it will become black.

If some vertex X is found connecting to a vertex S which color is gray during the process of iteration, then a cycle is found. The components of the discovered cycle is the sequence from S to X in the current DFS sequence.

Cycle-Check(G)

Input: A graph $\boldsymbol{G}$

Output: If cycle-detected, output the vertex sequence, otherwise, output 'No Cycle'.

```
for $u$ in $V$ do
|    $color[u]$ ← WHITE
|    $pred[u]$ ← NULL
end
for $u$ in $V$ do
|    if $color[u]$ $is\ equal\ to\ WHITE$ then
|    |    if DFSVisit($u$) $equal\ to\ 'STOP'$ then
|    |    |    return
|    |    end
|    end
end
return 'No Cycle'
```

DFSVisit(u)

Input: A vertex $\boldsymbol{u}$

Output: None

```
$color[u]$ ← GRAY
for $v \in Adj(u)$ do
|    if $color[v]$ $is\ equal\ to\ WHITE$ then
|    |    $pred[v]$ ← $u$
|    |    DFSVisit($v$)
|    end
|    if $color[v]$ $is\ equal\ to\ GRAY$ then
|    |    while $u\ is\ not\ equal\ to\ v$
|    |    |    print($u$)
|    |    |    $u = pred[u]$
|    |    end
|    |    print($v$)
|    |    return 'STOP'
|    end
end
$color[u]$ ← BLACK
return 'CONTINUE'
```

Time Complexity: O(V+E)