Problem 1

(a)

True, we can only do n-1 times of iteration in Bellman-Ford algorithm, if it is larger than this, it has negative cycle and it is not accurate.

(b)

False, it can find the shortest path.

(c)

True, this is the rule of Floyd's algorithm.

(d)

True, the Dijkstra's algorithm is similar to BFS because it uses the smallest visited node and it will generate a shortest path tree. So the time complexity is the same as BFS.

(e)

True, because except from O and D, the flow will be conserved. So the maximum flow will be determined by these two values.

(f)

True, because we will update the temporary distance when we scan the next node.

(g)

False, first we can not judge whether Prim or Kruskal is faster because we donnot know the number of edges and vertices. Second, the time of Kruskal is Nlog(N) but N is the number of edges not nodes.

(h)

True, the definition of the tree.

(i)

False, i and j will connect to other nodes, so the minimum weight connect to i and j can be that arc, not the arc between i and j.

(j)

True. These three are all greedy algorithms.

Problem 2

predecessor(p(j))	node	new distance
1	1	0
-	2	∞
-	3	∞
-	4	∞
-	5	∞
-	6	∞
-	7	∞
-	9	∞

predecessor(p(j))	node	new distance
1	1	0
1	2	1
1	3	4
1	4	7
-	5	∞
-	6	∞
1	7	6
-	9	∞

predecessor(p(j))	node	new distance
1	1	0
1	2	1
1	3	4
1	4	7
2	5	6
7	6	9
1	7	6
-	9	∞

predecessor(p(j))	node	new distance
1	1	0
1	2	1
1	3	4
1	4	7
2	5	6
7	6	9
1	7	6
6	9	18

predecessor(p(j))	node	new distance
1	1	0
1	2	1
1	3	4
1	4	7
2	5	6
7	6	9
1	7	6
6	9	18

predecessor(p(j))	node	new distance
1	1	0
1	2	1
1	3	4
1	4	7
2	5	6
7	6	9
1	7	6
6	9	18

It will not change, so this is the final version.

Problem 3

initial,

node	distance
1	0
2	∞
3	∞
4	∞
5	∞
6	∞
7	∞
8	∞
9	∞
10	∞
11	∞

visited 1,

node	distance
1	0
2	1
3	2
4	13
5	∞
6	∞
7	2
8	∞
9	∞
10	∞
11	∞

visited 1, 2,

node	distance
1	0
2	1
3	3
4	13
5	6
6	∞
7	2
8	∞
9	∞
10	∞
11	∞

visited 1, 2, 7,

node	distance
1	0
2	1
3	3
4	13
5	6
6	19
7	2
8	∞
9	∞
10	10
11	∞

visited 1, 2, 7, 3,

node	distance
1	0
2	1
3	3
4	13
5	6
6	14
7	2
8	∞
9	∞
10	10
11	∞

visited 1, 2, 7, 3, 5,

node	distance
1	0
2	1
3	3
4	13
5	6
6	10
7	2
8	20
9	21
10	10
11	∞

visited 1, 2, 7, 3, 5, 6,

node	distance
1	0
2	1
3	3
4	13
5	6
6	10
7	2
8	20
9	19
10	10
11	∞

visited 1, 2, 7, 3, 5, 6, 10

node	distance
1	0
2	1
3	3
4	13
5	6
6	10
7	2
8	20
9	18
10	10
11	17

visited 1, 2, 7, 3, 5, 6, 10, 4,

node	distance
1	0
2	1
3	3
4	13
5	6
6	10
7	2
8	20
9	18
10	10
11	17

visited 1, 2, 7, 3, 5, 6, 10, 4, 11,

node	distance
1	0
2	1
3	3
4	13
5	6
6	10
7	2
8	20
9	18
10	10
11	17

visited 1, 2, 7, 3, 5, 6, 10, 4, 11, 9,

node	distance
1	0
2	1
3	3
4	13
5	6
6	10
7	2
8	20
9	18
10	10
11	17

visited 1, 2, 7, 3, 5, 6, 10, 4, 11, 9, 8,

node	distance
1	0
2	1
3	3
4	13
5	6
6	10
7	2
8	20
9	18
10	10
11	17

Problem 4

Distance	1	2	3	4	Path	1	2	3	4
1	0	3	12	∞	1	-1	-1	-1	-1
2	∞	0	12	5	2	-1	-1	-1	-1
3	5	7	0	∞	3	-1	-1	-1	-1
4	∞	∞	3	0	4	-1	-1	-1	-1

use 1 as intermediate point,

Distance	1	2	3	4	Path	1	2	3	4
1	0	3	12	∞	1	-1	-1	-1	-1
2	∞	0	12	5	2	-1	-1	-1	-1
3	5	7	0	∞	3	-1	-1	-1	-1
4	∞	∞	3	0	4	-1	-1	-1	-1

use 2 as intermediate point,

Distance	1	2	3	4	Path	1	2	3	4
1	0	3	12	8	1	-1	-1	-1	2
2	∞	0	12	5	2	-1	-1	-1	-1
3	5	7	0	12	3	-1	-1	-1	2
4	∞	∞	3	0	4	-1	-1	-1	-1

use 3 as intermediate point,

Distance	1	2	3	4	Path	1	2	3	4
1	0	3	12	8	1	-1	-1	-1	2
2	17	0	12	5	2	3	-1	-1	-1
3	5	7	0	12	3	-1	-1	-1	2
4	8	10	3	0	4	3	3	-1	-1

use 4 as intermediate point,

Distance	1	2	3	4	Path	1	2	3	4
1	0	3	11	8	1	-1	-1	4	2
2	13	0	8	5	2	4	-1	4	-1
3	5	7	0	12	3	-1	-1	-1	2
4	8	10	3	0	4	3	3	-1	-1

Problem 5

First we need to initialize the distance from the original point n to other points and store these distances in an array dist[v], v is the number of vertices. The initial value from n to n will be 0 and other values will be infinity. And we need to add a counter to each vertex as counter[v].

From original point n, we find all edges (n,v) and based on the weight, we change the value of dist[v] if dist[n] + weight(n,v) ¡ dist[v]. If dist[n] + weight(n,v) = dist[v], we let counter[v] = counter[v] + 1. Else, we let counter[v] = 1. Then we mark n as visited.

After this, we find the minimum value of dist[v], we assume it as dist[k]. Also find all edges (k,v) and change the value if dist[k] + weight(k,v) ¡ dist[v]. Then we do the same as above. Mark k as visited and do this step again.

Until all vertices are visited, we can find the shortest path to destination d and it will store in dist[d]. And we need to multiply all counter[v] through this shortest path, the result is the number of shortest path.

Problem 6

Because you can't arrive earlier if you started later, the way is the same as Dijkstra's algorithm and it is a greedy algorithm. We just initial the time for all vertices and do it several steps we shown in Problem 5 and we can find the shortest path to each vertex. The counter part in Problem 5 can be cancelled because we donnot need it in this question.