Exercise 1

The original Floyd-warshall algorithm is like below.

G : given directed Graph

c : cost function which switches edge from vertex u to vertex v, (u, v) to c(u, v), which is the given cost of that edge. (cost is nonnegative from the exercise)

n : number of vertices in given graph, |V|

procedure Floyd-warshall(G,c)

for i:= 1 to n do

for j:=1 to n do

$d^0{}_{ij}$:= 0 if i=j

c(i,j) if i!=j and (i,j) ∈E

∞ otherwise

for k:= 1 to n do

for i:=1 to n do

for j:=1 to n do

$d^k{}_{ij}$:= min($d^{k-1}{}_{ij}$, $d^{k-1}{}_{ik} + d^{k-1}{}_{kj}$)

change of (min, +, cost set R) to

(a) (min, max, R)

change to

$d^k{}_{ij}$:= min($d^{k-1}{}_{ij}$, max($d^{k-1}{}_{ik}$, $d^{k-1}{}_{kj}$))

while every path from i to j, this algorithm computes smallest of the biggest edges in each paths from i to j.

when cost means the distance from i to j, and if every vertex is location of gas station, this algorithm shows the minimum longest distance without refueling when we need to move from i to j.

(b) (max, min, R)

change to

$d^k{}_{ij}$:= max($d^{k-1}{}_{ij}$, min($d^{k-1}{}_{ik}$, $d^{k-1}{}_{kj}$))

this algorithm computes the maximum value of minimum edge cost in the

path from i to j. we can say it with similar example from (a), by this algorithm, it computes the maximum shortest distance without refueling when we need to move from i to j(it can even be infinity when i to j needs at least more than 2 edges to traverse because unconnected(by an edge) pair of vertices is treated as infinity).

(c) (max, *, [0,1])
[0,1] scope of cost function can refer to probability of the edge passing, so changing min and + to max and * , we can get the path with the max probability. In real example we can think of something like mouse moving between points(vertices) when we cannot control but we know the probabilities of movements of mouse between vertices, the algorithm finds the most-possible probability of mouse's movement from a vertex to another.

(d) (+, *, {0,1})
change to
$d^k{}_{ij} := d^{k-1}{}_{ij} + (d^{k-1}{}_{ik} * d^{k-1}{}_{kj})$
will be number of paths from i to j.
from a map, this algorithm will compute number of paths between vertices.

Exercise 2

(a) making 200 Won by 10-, 50-, 100-Won coins
2*100Won
1*100Won + 2*50Won
1*100Won + 1*50Won + 5*10Won
1*100Won + 10*10Won
4*50Won
3*50Won + 5*10Won
2*50Won + 10*10Won

1*50Won + 15*10Won

20*10Won

there are 9 ways

(b)

Penny with value of $c_1, c_2, \dots, c_k$

As said in the exercise,

Suppose $w_{i,m}$ is the number of ways to change m pennies into coins with coins with values of $c_1, c_2, \dots, c_k$.

$w_{i,0} = 1, w_{0,m} = 0$ for all $i \in \{0, \dots, k\}$, $m \in \mathbb{N}$ and

$w_{i,m} = \begin{cases} w_{i,m-c_i} + w_{i-1,m} & if\ m \geq c_i \\ w_{i-1,m} & if\ m < c_i \end{cases}$ for $i \geq 1$.

When computing $w_{k,n}$

And for dynamic programming, assume

dp[n+1][k] : integer array for memorization

```
for (int i=0; i<=n; i++)
        for (int j=0; j<k; j++)
                if i==0,
                        dp[i][j]=1
                else if j==0,
                        if i%c_j==0,
                                dp[i][j]=1
                        else
                                dp[i][j]=0
                else if c_j > i,
                        dp[i][j]=dp[i][j-1]
                else
                        dp[i][j]=dp[i-c_j][j]+dp[i][j-1]
return dp[n][k-1]
```

this algorithm will count and return the number of the possible ways for coin change.

There is 2 overlapped for-loops but if k is treated as a constant, then,
The runtime of the resulting algorithm (finding $w_{k,n}$) = $O(n)$

(c)
there are 73681 number of ways for 2 Euros to be changed in given variation of coins.

Exercise 3

When Graph has n number of vertices ($n = |V|$)
And m number of edges ($m = |E|$)
The original prim algorithm is like below,

Prim's algorithm
Q : priority queue
D[v] : minimum cost of an edge {u,v} with u∈S,
S=V-Q
π[v] : "predecessor" of v
vertex u with D[v]=c(u,v)
T : growing tree

```
Procedure PRIM(G,c,s)                    G=(V,E)  graph
(1)   Q:=V-{s}                           C : E->R , S∈V
(2)   For all v∈V do D[v]:= 0            if v=s
(3)   (T:=s)            |  c(s,v)       if {s,v}∈E
                        |  ∞            otherwise
(4)   while Q≠ ∅ do
(5)       v:=deletemin(Q)    (one with minimum D-value)
(6)       add edge {π(v), v} to tree T
(7)       for all vertices u∈V adjacent to v do
(8)           if D[u]>c(v,u) then D[u]:=c(v,u) , π[u]:=v
```

the data structure for Q was heap to make priority queue, so the (5) and (8) took O(log n) runtime per execution.
But if the edge weights are in scope of integers from 1 to n, we can choose linked-list as priority queue then (5)deletemin becomes constant, and because we have only n variation of edge costs, and additionally we can keep addresses of upto n nodes which have D[v] of one of each 1 to n. (needs additional O(n) space)
Like
(head)vertices with D[v]=1 -- vertices with D[v]=2 -- vertices with D[v]=3 -- ...
/ / /
pointer pointer pointer
then (8)(changing the key value and moving it in linked-list)
needs O(log n) because we have the place address from 1 to n, we can divide-and-conquer to have O(log n).
Then,
For (1) O(n)
For (3) O(n)
Below (4) are executed n-1 times
For (5) constant
For (6) constant
➔ O(n)
Below (7) is O(m+n) (using adjacency list)
For (8) O(log n)
➔ O((m+n)logn)
Then, Time complexity = O(n) + O(n) + O(n) + O((m+n)logn) = O((m+n)logn)

When the edge weights are integers in the range from 1 to k for some constant k,
Use same data structure and the execution times are similar but (8) becomes constant because there are constant keys to compare, so

Time complexity = O(n) + O(n) + O(n) + O(m+n) = O(m+n)