> The `SCC` algorithm is the algorithm we used in the lecture that make a general directed graph in to a directed acrylic graph. For this problem specifically, notice that the eggs (weight) $w(v)$ of a new meta-node v is going to be the sum of the eggs of all the nodes which merged into new meta-node v. We know that this algorithm takes $O(V+E)$ times.
>
> The `MaxWeightPaths` algorithm is the algorithm developed in `GPS9.3`. It takes a `DAGs` as input, and outputs a `list[1..|V|]`, the `list[v]` is the maximum weight of a path starts from `v` in `DAGs`. (Notice this is slightly different than the one in `GPS9.3`). The algorithm itself takes $O(V+E)$ if we uses dynamic programing.
>
> We will take these two algorithms as predefined standard algorithms and won't prove their correctness here.

(a)

The pseudocode is list here. We use `SCC` first convert the `G` into a `DAGs` so we don't need to deal with cycles and could use `MaxWeightPaths` later. The problem of finding maximum number of eggs starting from s in G is then equivalent to finding maximum number of eggs starting from s' in $Dags$, where the s' is the meta-node that s merge into. Then the `MaxWeightPaths` gives us the maximum number of eggs we could collect for all possible path originated from s,this is the maximum number of eggs we could find if we start from s.

```
DAGs ← SCC(G)
eggs ← MaxWeightPaths(DAGs)

return eggs[s]
```

We see that running `SCC` and `MaxWeightPaths` only takes $O(V+E)$ so the total run time is $O(V+E)$

(b)

It's pretty much like part (a), but we need to consider the path start from any s now. This is done by calculating the `max` of all maximum path that originates from every possible s.

```
DAGs ← SCC(G)
eggs ← MaxWeightPaths(DAGs)

return max(eggs)
```

We see that running `SCC` and `MaxWeightPaths` only takes $O(V+E)$, the `max` takes $O(V)$, so the total run time is $O(V+E)$

(c)

We first use `SCC` to get `DAGs`, notice that each vertex v in `DAGs` has the information of which nodes in original G merges into it. We have a auxiliary function called `WeightOfNodes(v)` that gets all the nodes u in the original G and returns the weight of these nodes in a list. Just like (a), the problem of finding maximum number of eggs starting from any vertex in G given that only k nodes is chosen is

then equivalent to finding maximum number of eggs starting from any vertex in $Dags$ given that only k nodes is chosen from the meta-node in $Dags$.

The `maxN(list[1..n], k)` is a function that finds first `k` biggest elements in an array, and put them in an new array with size `k`. (If the list doesn't have `k` elements, it will fill the remaining part of the array with `0`s) It should be obvious that such function takes $O(nk)$ (one way to achieve this is to use `max` find the first maximum in the list, remove it, and then find the second maximum, so on and so forth, it's should be straightforward that this takes $O(nk)$). We see that if we compute `maxList` for every nodes, then it takes $\sum_i O(n_i k)$ and we know that the $\sum_i n_i = |V|$. (This basically means that sum of all the nodes in each meta-node is the number of vertex in G, which is trivial). Then we see that $\sum_i O(n_i k) = O(kV)$

The first part of the code is

```
DAGs ← SCC(G)

maxWeightOneNode[1..|V|, 0..k] ← []
maxList[1..|V|, 1..k] ← []

foreach v in DAGs:
    maxList[v] ← maxK(WeightOfNodes(v), k)

foreach v in DAGs:
    for i ← 0 to k:
        if i == 0:
            maxWeightOneNode[v, i] ← 0
        else:
            maxWeightOneNode[v, i] ← maxList[i] + maxWeightOneNode[v, i]
```

Then we have our first small DP. We have a memoization `maxWeightOneNode[1..|V|, 0..k]`, the first index is a vertex in `DAGs`, the second index is a number between `0` and `k`, inclusive. The `maxWeightOneNode[v, i]` means the maximum weight we could get if we choose at most `i` nodes from the v meta-node. It's not to hard to see that

$$\text{maxWeightOneNode}[v, i] = \begin{cases} 0 & i = 0 \\ \text{maxList}[v, i] + \text{maxWeightOneNode}[v, i-1] & i > 0 \end{cases}$$

We see that each case takes $O(1)$ time, and in total it thus takes $O(k)$ times for each meta-node in `DAGs`, and it takes $O(kV)$ to fill the table of $\text{maxWeightOneNode}[v, i]$. We need to iteration from `i = 0` to `i =k` so that each dependency will be defined when we need to use them. In this DP, we uses the `maxList`, to calculate `maxList[v, i]` we need $O(kV)$ as stated above. So, in total, the first part of the code takes $O(kV)$ time.

The second part of the code is:

```
maxWeight[1..|V|, 0..k] ← []
vertices ← TopologicalSort(DAGs)

for i ← 0 to k:
    foreach v in Reverse(vertices):
```

```
        maximum ← 0
        if |Out(v)| = 0:
            MaxWeight[v, i] ← maxWeightOneNode[v, i]
        else foreach u in Out(v):
            for j ← 0 to i:
                maximum ← max(maximum, maxWeightOneNode[v, j] + MaxWeight[u, i -
j])
            MaxWeight[v, i] ← maximum

return max(maxWeight)
```

This follows the similar structure of the algorithm used in `GPS9.3`. We use a memoization `maxWeight[1..|V|, 0..k]`, where `maxWeight[v, i]` means the maximum weight of all the path we could get given that we start from `v` in the `DAGs`, and we could choose at most `i` nodes in the all meta-nodes we go through in the path. We see that

$$\text{maxWeight}[v,i] = \begin{cases} \text{maxWeightOneNode}[v,i] & v \text{ is sink} \\ \max(\text{maxWeightOneNode}[u,j] + \text{MaxWeight}[u,k-j]) & \text{otherwise, forall out-neighbor } u \text{ and } j \in [0,i] \end{cases}$$

(If we are sink, then we couldn't go elsewhere, we just choose i nodes in current meta-node instead, otherwise, we could choose j nodes from ourselves, and $i-j$ from one of out-neighbor, then we try to find the maximum in each case)

If we see our recurrence, we see that to make sure each dependency will be defined when we need to use them, we need to iterate i from 0 to k, and j from 0 to k, and v through the reverse of topological order. The order of iterating neighbor doesn't matter.

Our answer should be the maximum of all the elements the 2-d `maxWeight` array (in fact, only the maximum of the last column `maxWeight [_, k]` will be okay, but this doesn't affect the overall time complexity), which will give the maximum weight of all the path starting from any vertex in `DAGs`, and we could choose at most `k` nodes in the all meta-nodes we go through in the path, and this is exactly what we want.

We see that `foreach v in Reverse(vertices) foreach u in Out(v), for j ← 0 to i` will be executed $O(kE)$ times in worst case, and each operation inside it is $O(1)$. So it takes $O(kE)$. We also see that the operation that is in `foreach v in Reverse(vertices)` but not in `foreach u in Out(v)` takes $O(V)$ times. So, all the staff in `foreach v in Reverse(vertices)` takes $O(kE+V)$ times, and we see that `for i ← 0 to k` takes $O(k)$ times, and therefore the total time complexity is $O(k^2E+kV)$. The `max` function in the end takes $O(kV)$ and the code from first part takes $O(kV)$ so in the end the total time complexity of the algorithm is that $O(k^2E+kV)$