Q 3.1)

When a graph is not acyclic, meaning there's at least one (directed) cycle. Since some calculations depend on the result of others, if there's a cycle, meaning that a calculation A needs to wait for calculation B to be computed but B is also waiting for A to be computed. Causing it to be impossible to compute any of them because they are forever waiting for each other.

So, it is impossible to perform all calculations in a cyclic graph (it is impossible to compute a group of calculations in a cycle).

Q 3.2)

Assuming the graph is an adjacency list. First having $pred(i)$, a set of computations that c_i depends on, and $s(i)$, the amount of time i^{th} vertex (c_i) takes to collate the result of computations that c_i depends on, all initialize as 0. By iterating through all m edges (all dependencies represent as $c_j \to c_i$), adding the amount of r_j into $s(i)$ representing $s(i) = \sum_{j \in pred(i)} r_j$, and adding index j into $pred(i)$ representing $pred(i) = \{j : (c_j \to c_i) \in E\}$, both take constant time.

Then adding a sink vertex into the graph and adding edges from all vertices with no outgoing edges to this sink vertex (no other calculations depend on these calculations). This represents the time requirement at the end which need to wait for the last among those vertices with no outgoing edges to finish computed. Set $s(n+1)$ to be 0 since this sink vertex isn't part of the calculations.

Subproblem: for all $1 \le i \le n+1$, let $P(i)$ be the problem of determining $opt(i)$, the minimum amount of time required to finish performing calculations up to c_i on the parallel computer.

Recurrence: for all $i > 0$,

$$opt(i) = \max\{opt(j) | \, j \in pred(i)\} + \begin{cases} 0, & if\ i = n+1 \\ {r_i}^2 + s(i), & otherwise. \end{cases}$$

c_i itself takes ${r_i}^2 + s(i)$ seconds to be computed (include collating the results of $pred(i)$), however, it still needs to wait until all $\{c_j | j \in pred(i)\}$ are computed, before c_i could start being computed.

By using the parallel computer to compute all those calculations c_i depends on at the same time in order to get the minimum amount of time require to perform up to c_i. Then only the maximum time require among those calculations is the time require for c_i to wait (since all other calculations c_i depends on would already be done computed because they take less time).

Since $opt(t)$ depends on all $\{opt(v) | v \in pred(t)\}$ for vertices v with outgoing edges to t, so we need to solve $P(v)$ for each such v before solving $P(t)$. This could be done by solving the vertices (calculations i to n and the sink vertex) in topological order, from left to right. All edges point from left to right, so any vertex with an outgoing edge to t is solved before t is.

And the sink vertex $(n+1)$ itself is not a calculation that need to be performed.

Base cases: for all $1 \leq k \leq n$ that $pred(k) = \emptyset$ then $opt(k) = {r_k}^2$.

Interpreting those calculations which depend on no others need to be computed first. Since we are using DP in topological order, those who are on the left most need to be computed first, so that those who are waiting for them to be computed could be computed next. This takes at most $O(n)$ time for iterating through n vertices.

$opt(n+1)$ (Sink vertex), is the minimum amount of time required to perform all n calculations on the parallel computer, because we need to wait until the last calculation is computed, and since all of its left vertices from topological order need to be done computing first.

Overall time complexity is $O(n+m)$. Iterating through m edges in $O(m)$ time. By using topological sort in $O(n+m)$ time having n vertices (calculations) and m edges. Then adding a sink vertex and edges in $O(n)$ times for at most n edges from n calculations (vertices). Initializing base cases in $O(n)$ time. For the recurrence, each edge (dependency) is only considered once (at its end point) costing $O(m)$ time (tighter bound). Costing $O(3n+3m) = O(n+m)$ in total.

Q 3.3)

Setting up $s(i)$, sink node and do a topological sort just like in Q3.2 (with the same reasons) in $O(n+m)$ time.

Subproblems: for all $1 \leq i \leq n+1$ and $1 \leq j \leq s$, let $P(i,j)$ be the problem of determining $opt(i,j)$, the minimum amount of time require to finish performing calculations up to c_i using the supercomputer for at most j calculations, and the parallel computer for all other calculations.

Recurrence: for $i > 0$ and $1 \leq j \leq s$,

$$opt(i,0) = \max\{opt(k,0) \mid k \in pred(i)\} + \begin{cases} 0, & if\ i = n+1 \\ {r_i}^2 + s(i), & otherwise. \end{cases}$$

This is exactly like Q3.2 for the case of not using any super computers.

$$opt(i,j) = \min\begin{pmatrix} opt(i,0) - {r_i}^2, \\ \max\{opt(k,j-1) \mid k \in pred(i)\} + {r_i}^2 \end{pmatrix} + s(i)$$

$$opt(n+1,j) = \max\{opt(k,j-1) \mid k \in pred(i)\}$$

Interpret choosing a minimal choice between using super computer at current calculation or had used in the past of current one's dependencies.

And the sink vertex $(n+1)$ itself is not a calculation that need to be performed.

Since $opt(t,j)$ depends on all $\{opt(v,j-1)|v \in pred(t)\}$ for vertices v with outgoing edges to t, so we need to solve $P(v,j-1)$ for each such v before solving $P(t,j)$. This could be done by solving the vertices (calculations i to n and the sink vertex) in topological order, from left to right. All edges point from left to right, so any vertex with an outgoing edge to t is solved before t is. Then increasing j.

Base cases: for all $1 \leq l \leq n$ that $pred(l) = \emptyset$ then $opt(l,0) = r_l{}^2$.
With same reasons as Q3.2.

$opt(n+1,s)$ (Sink vertex), is the minimum amount of time required to perform all n calculations on the parallel computer using at most s times of super computer, because we need to wait until the last calculation is computed, and since all of its left vertices from topological order need to be done computing first.

Overall time complexity is $O(s(n+m))$. Taking $O(n+m)$ for setting up, and using similar logic as Q3.2 but repeating it s times causing it to be $O(s(n+m))$. $O\big(n+m+s(n+m)\big) = O(s(n+m))$ in total.