Question 1

Define $Opt(k)$ to be the minimum cost plan for purchasing apples during the first $k$ weeks, where we do not allow for a purchase plan that extends beyond the $k$th week. In other words, we do not want to have a 4-week contract for apples from a wholesaler to start on any of the last 3 of these $k$ weeks. By this definition, $Opt(n)$ is the value we are looking for.

The key observation in calculating $Opt(k)$ is that we should either end with a purchase from the local farmers, and use the optimal solution for the first $k-1$ weeks, or we should just be finishing a 4-week purchase from a wholesaler, and use the optimal solution for the first $k-4$ weeks. Note that this recurrence only makes sense for $k \geq 4$.

Therefore we have that

$$Opt(k) = \min\{c_k + Opt(k-1), c' + Opt(k-4)\}$$

for $k \geq 4$. As base cases, $Opt(0) = 0$, $Opt(1) = c_1$, $Opt(2) = c_1 + c_2$ and $Opt(3) = c_1 + c_2 + c_3$. Since we would like to know the actual purchase plan, and not just the lowest possible cost, we will also store a solution for all sub-problems.

Therefore a correct algorithm is as follows:

```
AppleScheduler
S[0...n] // Where we will record Opt
T[0...n] // Where we will record the corresponding schedule
S[0] = 0, T[0] = 0 // base cases
S[1] = c_1, T[1] = (L)
for k = 4 to n
  if c_k + S[k-1] \leq c' + S[k-4]
    S[k] = c_k + S[k-1]
    T[k] = T[k-1]; L
```
else
    \[ S[k] = c' + S[k - 4] \]
    \[ T[k] = T[k - 4] : W \]
endfor
return(\(T[n]\))

Since each sub-problem can be computed in constant time from the previous sub-problems, all \(n\) can be solved in linear time. The running time of this algorithm is \(O(n)\). Note that this algorithm uses \(O(n^2)\) space. We could have solved the problem in \(O(n)\) space by not carrying around full solutions, but rather constructing them backwards from \(S[k]\) at the end.

**Question 2**

A natural sub-problem here is similar in spirit to that of Question 1. In particular, we can define \(Opt(k)\) to be the cheapest means of carrying out the first \(k\) steps in the production process. We seek the value \(Opt(n)\).

To determine the value of \(Opt(k)\), first note that the \(k\)th task must be done at some lab, but it might be done at a single lab in sequence with any number of earlier tasks. In other words, if in the optimal solution task \(k\) is done at some lab \(L\), then it is possible that all tasks were done at lab \(L\), or just tasks 2 through \(k\), or tasks 3 through \(k\), and so on. In the final case, the optimal solution was at a different lab just prior to doing task \(k\). Therefore we have the recurrence

\[
Opt(k) = \min_{0 \leq i \leq k-1} \{ f(i + 1, k) + Opt(i) + c \}
\]

where we define \(Opt(0) = 0\). Here is a suitable algorithm:

**ComputerMaker**

\[ S[0...n] \] // Where we will record \(Opt\)
\[ S[0] = 0 \]
for \(k = 1\) to \(n\)
    \[ S[k] = \min_{0 \leq i \leq k-1} \{ f(i + 1, k) + S[i] + c \} \]
endfor
return(\(S[n]\))

Note that the problem did not ask for the actual building plan, just the cost, so this solution is sufficient. Also note that as in many of these problems, there is more than one way to handle the base cases.
Each sub-problem is calculated using the min of all previous sub-problems. Therefore each entry in our array is calculated in $O(n)$ time, for a total running time of $O(n^2)$.

**Question 3**

a) If $n = 3$, $k = 1$ and we have $r_1 = 4$, $r_2 = 6$, $r_3 = 4$ then the maximum weight independent set is $\{1, 3\}$, with a weight of 8. Therefore the suggested algorithm will take one of these, and return the value 4. However, the optimal solution consisting of 1 station clearly uses station 2, for a weight of 6.

b) The hardest part here is coming up with a good sub-problem. Define $Opt(m, i)$ to be the greatest attainable revenue if we are allowed to open up to $i$ stations on any independent set of the first $m$ rest areas. Note that we say “up to $i$ stations,” since sometime a larger profit can be made if we are allowed to use fewer stations. (Can you think of a simple example where this is the case?) Under our notation, we are interested in finding $Opt(n, k)$.

Our goal will be to define $Opt(m, i)$ in terms of smaller sub-problems. If all goes as planned, we should only have to consider $n \cdot k$ of these sub-problems.

Let’s put off worrying about the base cases, and see what we can say about $Opt(m, i)$. The observation we used in class, which will be useful here too, is that either $Opt(m, i)$ builds a station at location $m$, or it does not. If it does not build at $m$, then we have an independent set of at most $i$ stations built on rest areas 1 through $m - 1$. Therefore $Opt(m, i) = Opt(m - 1, i)$.

Alternatively, $Opt(m, i)$ does build at the $m^{th}$ rest area. In that case, $Opt(m, i)$ cannot build at rest area $m - 1$. Furthermore, on rest areas 1 through $m - 2$, $Opt(m, i)$ can build at most $i - 1$ stations, since otherwise we would be using more than $i$ stations altogether for $Opt(m, i)$. Since we gain $r_m$ for building a station at location $m$, $Opt(m, i) = r_m + Opt(m - 2, i - 1)$.

We can now put these together, creating a recurrence. Since we are trying to maximize our revenue, we have that

$$Opt(m, i) = \max\{Opt(m - 1, i), r_m + Opt(m - 2, i - 1)\}.$$
As base cases, note that \( \text{Opt}(m, 0) = 0 \) for all \( m \), and \( \text{Opt}(0, i) = 0 \) for all \( i \). These alone could lead to problems, as our recurrence calls for an \( \text{Opt}(m - 2, i - 1) \), and therefore when we calculate \( \text{Opt}(1, 1) \) we will try to look up a value for \( \text{Opt}(-1, 1) \). One solution is to note that \( \text{Opt}(1, i) = r_1 \) for all \( i > 0 \). (This assumes that \( r_1 \geq 0 \). Without this assumption we would need \( \text{Opt}(1, i) = \max\{r_1, 0\} \).) Again, we note that there are multiple ways to handle the base cases for this problem. Here is an algorithm putting it all together:

\begin{verbatim}
GasStations
    S[0...n, 0...k]   // Where we will record Opt
    S[m, 0] = 0 for all m // Base cases
    S[0, i] = 0 for all i
    S[1, i] = r_1 for all i > 0
    for m = 1 to n
        for i = 1 to k
            S[m, i] = max{S[m - 1, i], r_m + S[m - 2, i - 1]}
        endfor
    endfor
    return(S[n, k])
\end{verbatim}

Each entry in our table can be calculated in constant time, as each requires that we look up only two values (which we have already computed), and then take the maximum of two terms. This is done \( nk \) times, and this dominates all other work done in the algorithm. Therefore, the running time is \( O(nk) \).