Question 1 (Karatsuba Multiplication, 30 points). What three recursive top-level multiplies are called for when one uses Karatsuba multiplication to multiply the binary numbers 10110101 and 11101001?

Please write out the three new multiplication calls (you don’t need to evaluate the answers). Your answers should also be in binary.

These numbers can be decomposed as $10110101 = 1011X + 0101$ and $11101001 = 1110X + 1001$ where $X = 2^4$. The recursive calls are then $1011\cdot1110$, $0101\cdot1001$ and $(1011+0101)(1110+1001) = 10000\cdot10111$. 
**Question 2** (Independent Set Computation, 30 points). *Compute the size of the largest independent set of the graph given below.*

![Diagram of a tree structure with node values labeled from 1 to 13.]

*It is recommended that you show your work so that you can be given partial credit if your answer is not correct.*

The answer is 13 based on the above numbering. These are computed using the dynamic program for independent set in trees, where the maximum size of an independent set of a node’s subtree is the maximum of the sum of this quantity over its children and one more than the sum over its grandchildren.
**Question 3** (Vacation Days Planning, 35 points). Over the course of the next \( n \) workdays Abby accrues one vacation day every 30 (even if she is on vacation at the time). She can save up vacation days as much as she likes, but can never spend days that she hasn’t earned yet. During this period, there are a number of possible vacations that she is considering going on. Each vacation has a range of consecutive days that it would take. Abby cannot go for only part of the vacation, nor can she go on more than one vacation at once. Each vacation also has a fun rating.

Abby wants to figure out which collection of vacations she should go on in order to maximize her total fun without exceeding her allotted vacation days. Give a polynomial time algorithm to determine the most possible fun she can achieve and analyze its runtime.

[Hint: for each day and each \( k \) find the most amount of fun that Abby can have by that day with at least \( k \) vacation days remaining.]

We assume that the date range consists of \( d \) days labelled 1, 2, \ldots, \( d \) with Abby getting another vacation day on each day that is a multiple of 30. We note that she can never collect more than \( d/30 \) vacation days.

We let \( \text{Best}(y, k) \) be the most total fun she can obtain by day \( y \) while still having \( k \) vacation days left and not being in the middle of a trip. We note that this is the maximum of \( \text{Best}(y - 1, k) \) (or \( \text{Best}(y - 1, k - 1) \) if \( y \) is a multiple of 30) and the maximum of \( \text{Best}(y', k' + k') + f \) over trips \( t \) which run from day \( y' \) to \( y \) with fun value \( f \) that take up \( k' \) vacation days (accounting for accrued vacation days during the interim).

The algorithm is as follows:

**BestTripValue**

Create \((d) \times (d/30 + 1)\) array \( A \) with all entries initialized to -Infinity

For \( y = 1 \) to \( d \)

For \( k = 0 \) to \( d/30 \)

If \( y = 1 \) and \( k = 0 \)

\( A[y, k] \leftarrow 0 \)

Else

If \( y \) is a multiple of 30

\( A[y, k] \leftarrow A[y-1, k-1] \)

Else

\( A[y, k] \leftarrow A[y-1, k] \)

For each trip \( T \) ending at day \( y \)

Let \( T \) start at \( y' \), and let \( k' \) be \( y - y' \) (the number of multiples of 30 between \( y \) and \( y' \)) with fun \( f \)

\( A[y, k] \leftarrow \max(A[y', k+k'] + f, A[y, k]) \)

Return maximum over \( k \) of \( A[d, k] \)

Correctness depends on noting that our recurrence for the best value function is correct. The best schedule that can leave \( k \) vacation days at day \( y \) must either have a trip ending at day \( y \) or not. In the latter case, Abby is following a schedule ending day \( y - 1 \) with \( k \) (or \( k - 1 \) if \( k \) is a multiple of 30) vacation days, and has value at most \( \text{Best}(y - 1, k) \) (or \( \text{Best}(y - 1, k - 1) \)). In the case that she just got back from a trip \( T \) starting at \( y \), she must have had \( k + k' \) vacation days at \( y' \) and the best value she could get would be \( \text{Best}(y', k + k') + f \). Taking the best of these options will give the correct value of \( \text{Best}(y, k) \). Since our algorithm will have previously computed the correct value of \( \text{Best}(y', k + k') \) for all such trips, it will correctly compute each value of \( \text{Best}(y, k) \), and the final output will be the maximum of \( \text{Best}(d, k) \) over all \( k \).

Runtime is easily seen to be polynomial. The loops over \( y \) and \( k \) iterate over \( O(d^2) \) possibilities. For each one, one then needs to iterate over all trips. The final runtime is at most \( O(d^2 \text{(number of trips)}) \).
**Question 4** (Travel Planning, 35 points). Wendy is trying to travel through Neverland. She has a map of locations and routes including how long each route takes to traverse. Unfortunately, the landscape is constantly changing, and each route will only be passible for a known interval of time (though she can wait as long as she wants at any of the locations).

Wendy wants an algorithm that given a time for the start of her journey, $n$ locations (two designated as start and end), and $m$ routes where for each route she is given the start location of the route, the end location of the route, the time it takes to traverse, and the start and end times of its window of availability (she can only begin her journey along the route at times during the window).

Give an algorithm that allows her to compute the earlier possible time that she could arrive at her destination. For full credit, your algorithm should run in time $O(n \log(n) + m)$ or better.

Our solution is modeled off of Dijkstra’s algorithm. For each location we attempt to compute the earliest time that it can be reached. We do this by maintaining for each location $x$ a time $t(x)$, the earliest time of arrival for any known route to that location from our starting location. We maintain a queue of locations whose arrival time has not been finalized yet. For each other $x$, $t(x)$ will denote the earliest time of arrival to $x$ from any valid route that otherwise passes only through locations whose best arrival time is known. We note that this can be computed by considering only all paths that head from a location whose best time is known to one whose best time isn’t. This can be easily updated when a new location is added to the list of those with known times.

We also note that the location with the smallest value of $t(x)$ will have $t(x)$ as its actual best arrival time. This is because any path to $x$ must first reach some vertex with unknown best arrival time (which takes time at least $t(x)$), and then potentially take longer. The algorithm is as follows:

```plaintext
BestTravel(Locations, Routes, StartLocation, EndLocation)

Scan through routes and compile for each location a list of routes that start at that location
For each location $x$ set $t(x) \leftarrow \infty$
$t(StartLocation) \leftarrow StartTime$
Create Priority Queue $Q$
Insert all locations $x$ into $Q$ sorted by $t(x)$
While $Q$ non-empty
    $u \leftarrow \text{DeleteMin}(Q)$
    For each route $R$ starting at $u$
        Let $w$ be the endpoint of $R$
        If the end of $R$’s availability is after $t(u)$
            $\text{FinishTime} \leftarrow \text{TravelTime}(R) + \max(t(u), \text{Start of R’s availability})$
            If $\text{FinishTime} < t(w)$
                $t(w) \leftarrow \text{FinishTime}$
                $\text{DecreaseKey}(w)$
        Return $t(EndLocation)$
```

To see that the updates are correct, we note that $t(w)$ should only change if there is a new route to $w$ through $u$. This would happen only if there was a route from $u$ to $w$, and only if this route’s availability expires only after $t(u)$. If the route is available, one could leave from $u$ to $w$ at the later of $t(u)$ or the start of the window of availability. One would obtain the time of arrival at $w$ by adding the trip length to this, and $t(w)$ should be replaced by the minimum of its current value with this new one.

For the runtime analysis, one notes that the runtime is dominated by the $m$ decrease key operations and the $n$ insert and delete minimum operations. If one implements the priority queue with a Fibonacci Heap, this can be done in $O(n \log(n) + m)$ time.
**Question 5** (Bad Chain Avoidance, 35 points). Let $G$ be a DAG whose vertices are labelled black and white. Give an algorithm that given $G$, an integer $k$ and vertices $s$ and $t$ determines whether or not there is a path from $s$ to $t$ that does not pass through $k$ black vertices in a row at any point.

For full credit your algorithm should have runtime $O(|V| + |E|)$ or better.

For each vertex $v$ we will compute $\text{short}(v)$ which is $k$ if there is no path from $s$ to $v$ without $k$ consecutive black vertices and otherwise is the minimum over such paths of the number of trailing black vertices. It is easy to see that $\text{short}(s)$ is 1 if $s$ is black and 0 otherwise.

Otherwise, $s(w)$ if $w$ is white is 0 if there is an edge from $u$ to $w$ with $\text{short}(u) < k$ and $k$ otherwise. This is because if there is such a $u$, combining the appropriate path with the edge $(u, w)$ gives a valid path to $w$, which now ends in a white vertex, which gives $\text{short}(w) = 0$. Otherwise, there is no such path so $\text{short}(w) = k$.

For $w$ black, we have that $\text{short}(w)$ is one more than the minimum of $\text{short}(u)$ over edges $(u, w)$ (or $k$ if this is larger). This is because any path leading to $w$ must first go to some such $u$ and be followed by the edge $(u, w)$. This will have one extra trailing black vertex than the previous path.

The algorithm is now as follows:

Path($G$)
Topologically Sort $G$
For $w$ in $G$ in topological order
  If $w = s$
    If $w$ is white
      $\text{short}(w) = 0$
    Else
      $\text{short}(w) = 1$
  Else
    Let $m =$ minimum over edges $(u, w)$ of $\text{short}(u)$
    If $m = k$
      $\text{short}(w) = k$
    Else If $w$ is white
      $\text{short}(w) = 0$
    Else
      $\text{short}(w) = m+1$
  If $\text{short}(t) < k$
    Return True
  Else
    Return False

Correctness follows from the fact that the recurrence for $\text{short}(w)$ described above is correct. The fact that we topologically sorted $G$ means that when trying to define $\text{short}(w)$ we have already (correctly) computed $\text{short}(u)$ for all edges $(u, w)$.

For runtime, the topological sort is linear time, and additionally, we need to iterate over each edge and vertex, for a final runtime of $O(|V| + |E|)$. 
**Question 6** (Greedy Candyland, 35 points). In the game of Candyland, the board consists of a path of colored squares. You begin at the first square and try to reach the end of the path. On each turn you draw a card with a color on it and advance your piece to the next square with that color (or stay where you are if there are no further squares of that color).

Adam is able to cheat by stacking the deck in order to get whatever colors of cards he wants. In order to get to the end in as few moves as possible, he arranges the deck so that on each turn he gets the color of card that will allow him to advance his piece as far as possible.

Does this necessarily allow Adam to finish the game in the minimal number of turns? Prove this or provide a counterexample.

Let the greedy algorithm reach a sequence of squares $g_1, g_2, \ldots, g_k$. Consider any other possible sequence of squares $a_1, a_2, \ldots, a_m$. We claim by induction on $n$ that $g_n \geq a_n$ (i.e. that the square $g_n$ is at least as far along on the path). For $n = 1$, this is clear since $g_1$ and $a_1$ are both the starting square.

Next assume that $g_n \geq a_n$, we wish to show that $g_{n+1} \geq a_{n+1}$. Suppose that $a_{n+1}$ was reached from $a_n$ by drawing a red card. This means that $a_{n+1}$ is the first red square after $a_n$. However, $g_n \geq a_n$. This means that the first red square after $g_n$ is at least as far along as $a_{n+1}$. Thus, drawing a red card from $g_n$ gets at least as far as $a_{n+1}$. Since $g_{n+1}$ is the further you could get from $g_n$ in one step, $g_{n+1} \geq a_{n+1}$. Thus, by induction $g_n \geq a_n$ for all $n \leq \min(m, k)$. This means that the greedy solution must require fewer steps than the arbitrary one. In particular, if $k > m$, then $g_m \geq a_m$ so the greedy solution would have finished at that time, a contradiction.