1. **Solution:**
   - The recursion tree for this equation is lopsided. The sums of the complete levels in the tree form a descending geometric series $n + \lfloor \frac{5}{6} \rfloor n + \lfloor \frac{5^2}{6^2} \rfloor n + \ldots$. For an upper bound, we ignore that the tree ends and take the infinite sum of this series, whch gives the bound $T(n) \leq \frac{1}{1-5/6} n = O(n)$. For the lower bound, we consider the sums of only the complete levels in the recursion tree, which is again dominated by the sum of the first level. This gives the bound $T(n) \geq n = \Omega(n)$. Together they bound the recurrence as $T(n) = \Theta(n)$.
   - The figure 1 shows the corresponding DFA. The states are labelled nc, where n is a number from $\{0, 1, 2\}$, and b is a letter from $\{d, e\}$. The number signifies the minimum number of 0s any string ending at the state has, and the letter signifies if the length of the string is even (e) or odd (d) (the number 0 and letter o may look the same, hence 'odd' is denoted by 'd' instead). For example, state $1e$ denotes the set of strings that contain at least one 0, and have even length. State $0e$ is the start state, and as required, $2e$ is the only accept state for the language accepting strings that contain at least two 0s and have even length.

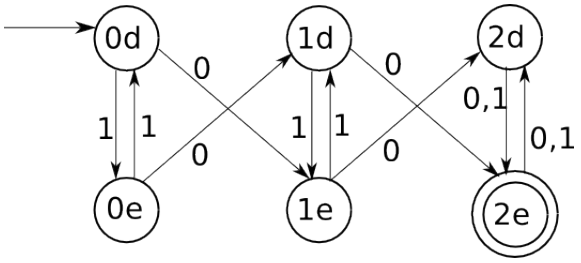

Figure 1. DFA for language that has at least two 0s and even length strings

■

2. (a) $\{u, v, w, t\}$
   (b) Define graph $G_\varepsilon = (V, E)$ where $V = Q$ and $E = \{u \to v \mid \delta(u, \varepsilon) = v\}$ for some $u, v \in Q$. Define the ε-reach $\delta^*(v, \varepsilon)$ as the set of states reachable from v using only ε transitions. This set can be found by performing BFS or DFS on G_ε starting at v. Finding the ε-reach from one state takes at most $O(n + m)$ time.
   Starting at q, let set $X = \delta^*(q, \varepsilon)$. Define set $W = \{u \mid \exists x \in X : u = \delta(x, a)\}$. Finding W takes $O(n + m)$ time because we have to consider each transition at most once.
   Finally, define $G'_\varepsilon = (V', E')$ where $V' = V \cup \{s\}$ and $E' = E \cup \{s \to w \mid w \in W\}$
   $\delta^*(q, a)$ is the set of vertices visited when performing BFS or DFS on G'_ε starting at s. This will end up taking $O(n + m)$ time.
   The total running time is $O(n + m)$.

3. (a) There are two possible solutions, one using a modified Prim's algorithm, and one using connected components.

   Solution (Modified Prim's): The idea behind this is that Prim's grows the spanning tree by maintaining a tree and adding one vertex to the tree at a time. This works in linear time for us, because the choice of edge is made efficient without the need to sort. First construct two bags containing 1-weight edges and 25-weight edges respectively . It takes $O(E)$ to build the lists, The find operation is $O(1)$, the decrease key operation is $O(1)$. Thus, the algorithm runs in $\boldsymbol{O(V + E)}$ ***time***. ■

Solution (Connected Components): Find the connected components in the graph with the smallest weight set of edges. Shrink them and find the connected components again with edges of next weight, shrink them, and continue this procedure until we find a connected graph or exhaust the set of edges. This algorithm also runs in ***O(V + E) time*** since we only have 2 distinct edge weights and finding connected components and shrinking takes linear time. ■

4. **Solution:** We solve this problem by DP. Let $n := |w|$. Let MinLInf$[i]$ denote the minimum ℓ_∞ measure of a valid split of $w[i \dots n]$ with respect to L if one exists. If there is no valid split for $w[i \dots n]$, then MinLInf$[i]$ should be ∞. Otherwise, for any j such that $w[i \dots j]$ a string in L, we can make a first split after index j. Then the ℓ_∞ measure of this split of $w[i \dots n]$ is the maximum of the length of $w[i \dots j]$ and the ℓ_∞ measure of a valid split of the substring $w[j+1 \dots n]$. Among all valid first splits we take the one with minimum ℓ_∞ measure. Thus, we have the following recursion.

$$\text{MinLInf}[i] = \min_{j \geq i : \text{IsStringInL}(w[i \dots j])=1} \max\{(j-i+1), \text{MinLInf}[j+1]\}$$

For the base case, we set MinLInf$[n+1] = 0$. We can use an array to store the values of MinLInf$[i]$ for all $i \in \{1, \dots, n+1\}$. We initialize MinLInf$[i] = \infty$ for all $i \in \{1, \dots, n\}$ and MinLInf$[n+1] = 0$. We compute the entries of the array in decreasing order of indices. After filling the whole array, we return the value of MinLInf$[1]$. The above algorithn can be summarized in the following pseudocode.

```
MinLInf(w, IsStringInL(·)):
  initiate MinLInf to be an array of size n + 1, where n = |w|
  MinLInf[n + 1] = 0
  for i from n to 1:
      MinLInf[i] = ∞
      for j from i to n:
          if IsStringInL(w[i ... j]) = 1:
              MinLInf[i] = min{MinLInf[i], max{(j − i + 1), MinLInf[j + 1]}}
  return MinLInf[1]
```

To compute MinLInf$[i]$ for each $i \in [n]$, we need $O(n)$ time as we need to iterate through all $j \geq i$ to check if $w[i \dots j] \in L$ and for each j such that $w[i \dots j] \in L$, we do constant extra work of reading value from the array, algebra, and comparison. Since the array has $O(n)$ size and it takes $O(n)$ time to compute each entry of the array, the algorithm runs in $O(n^2)$ time in total. ■

5. A kite of size k is a complete graph (clique) on k vertices plus a tail of k vertices. See figure for a kite of size 4. The KITE problem is the following: given an undirected graph $G = (V, E)$ and an integer k does G contain a kite of size k as a subgraph? Prove that KITE is NP-Complete.

   To show that KITE is NP-Complete we need to show that it is NP-Hard and also that it is in NP.

   Solution: NP-Hard: We reduce CLIQUE to KITE to show that KITE is NP-Hard. Consider an instance of CLIQUE, which is a graph $G = (V, E)$ and an integer k. We reduce this instance to the following instance of KITE. For every node

$u \in V(G)$, we add a path of $k-1$ new vertices, which we denote by P_u. The resulting graph G' has

$$V(G') = V(G) \cup \left(\bigcup_{u \in V(G)} V(P_u) \right) \text{ and } E(G') = E(G) \cup \left(\bigcup_{u \in V(G)} E(P_u) \right)$$

We can create G' in polynomial time with respect to the input, by iterating over every vertex $u \in V(G)$ and performing the following process. We first create the path P_u with $k-1$ new vertices. Finally, we add an edge from one of the endpoints of P_u to u. Next, we need to show that this reduction is valid. In other words, we need to prove the following claim.

Claim 1. *G' has a kite of size k if and only if G has a clique of size k.*

Proof:
* $\Longrightarrow$ Suppose that G' has a kite of size k. Let S be the set of vertices in the kite that form a clique of size k. Notice that any vertex in S is also a vertex in $V(G)$ or, in other words, no vertex in S was added to G when we created G'. To see why that is true, notice that, for any $u \in V(G)$, any vertex $v \in P_u$ that we added when we created G' from G had degree either 2 or 1 as it belongs to P_u which is a path. Therefore, for $k > 2$, no such vertex can belong in a clique in G', which implies that no such vertex can belong in S. Therefore, $S \subseteq V(G)$. Thus G has a clique of size k, the clique formed by the vertices in S.
* $\Longleftarrow$ Suppose now that G has a clique of size k. Let S be the set of vertices that form this clique, and let u be an arbitrary vertex in S. Then, $S \cup P_u$ is a kite in G', as S is a clique with k vertices, $P_u \cup \{u\}$ is a path with k vertices, and $S \cap (P_u \cup \{u\}) = \{u\}$. Thus, $S \cup P_u$ is a kite of size k in G'.

□

Thus we have shown that CLIQUE reduces to KITE in polynomial time, which implies that KITE is NP-Hard.

– In NP: We show that a verifier can, in polynomial time, verify whether a potential solution is actually a solution or not. A certificate for KITE is two sets S and T that correspond to the set of vertices that form a clique of size k and the set of vertices that form a path of size k, respectively. The verifier can, in polynomial time:
  * Verify that $S \subset V(G)$ and $T \subset V(G)$, by checking that any vertex in $S \cup T$ is also in $V(G)$.
  * Verify that $|S| = k$ and $|T| = k$, by iterating over all vertices in S and T.
  * Verify that the vertices of S forms a clique by checking to see that for any pair of vertices $x, y \in S$, the edge $\{x, y\}$ exists.
  * Verify that $S \cap T$ contains exactly one vertex, which we denote here as u.
  * Verify that the set of vertices of T forms a path by starting from the vertex $v \in T$ that has degree exactly 1 (one endpoint of the path) and traversing the path until they find u, making sure that intermediate vertices have degree exactly 2.

  If all the above hold, then the verifier decides that the certificate is valid, otherwise it decides that it is not. Thus, we have shown that a polynomial-time verifier can decide whether a certificate is actually a solution or not, which implies that KITE is in NP.

Since we showed that KITE is NP-Hard and is also in NP, we conclude that KITE is NP-Complete.

6. (a) **Solution:** (Proof by contradiction.) Suppose there exists an algorithm DecideL that correctly decides the language L. Then we can solve the halting problem via the following algorithm.

```
DecideHalt(⟨M, w⟩):
  Encode the following Turing machine M′:
      M′(x):
        run M on input w
        return TRUE
  if DecideL(⟨M′⟩)
      return TRUE
  else
      return FALSE
```

To prove correctness, we show that M halts on w if and only if DecideHalt accepts the encoding $\langle M, w\rangle$.

First, assume M halts on w. Then M' accepts every input string x. In particular, M' accepts at least one string. Thus, DecideL accepts $\langle M'\rangle$, implying that DecideHalt accepts $\langle M, w\rangle$.

Now assume M does not halt on w. Then M' diverges on every input string x. In particular, M' does not accept any strings. Thus, DecideL rejects $\langle M'\rangle$, implying that DecideHalt rejects $\langle M, w\rangle$.

Therefore, DecideHalt correctly decides the halting problem, which is a contradiction since we know the halting problem is undecidable. Thus, no such algorithm to decide L exists. ■

(b) **Solution:** We describe a Turing machine M' that accepts the language L. Let $s_1, s_2, s_3, \ldots$ be the enumeration of strings in Σ^* in lexicographic order (or any other computable ordering of the strings).

```
M′(⟨M⟩):
  for k ← 1 to ∞
      for i ← 1 to k:
          run M for k steps on input s_i
          if M accepts s_i
              return TRUE
```

To prove correctness, we show that M' accepts $\langle M\rangle$ if and only if M accepts at least one string.

First, assume that M' accepts $\langle M\rangle$. Then there exists i such that M accepts s_i. Therefore, M accepts at least one string.

Now assume M accepts at least one string, denoted as s_j. As M accepts s_j, there exists a finite number of steps k that M executes on input s_j to reach an accepting state. Then after $\max\{k, j\}$ iterations of the outer for loop of M', M' will run M on input s_j for at least k steps. Thus, M' will accept $\langle M\rangle$.

Therefore, M' accepts the language L. ■

7. **Solution:** We can come up with a greedy algorithm for this problem. The pseudocode for the algorithm has been outlined here.

```
NumIntervals(S):
  count = 0        # The minimum number of intervals required
  maxPos = -inf    # The farthest position covered by our strategy
  for i in [1...len(S)]:
      if x_i > maxPos:
          # Add an interval [x_i, x_1 + 1]
          maxPos = x_i+1
          count = count + 1
  return count
```

In the above pseudocode, `maxPos` keeps track of the maximum position to the right of the real line which is covered by intervals chosen so far and `count` stores the number of intervals chosen so far. Whenever the current point, say x_i is not covered by intervals so far, we add $[x_i, x_i+1]$ as a new interval.
This algorithm would have a runtime of $O(n)$, where n is the length of array S. This is easy to see, because the pseudocode above has just the one for loop over S and all the computations done can be performed in $O(1)$ time otherwise.

We now prove that this strategy would be an optimal strategy. [NOTE: This proof is not required as part of the answer in the examination, but is being provided here for the sake of completion].

Claim 1: There is always an optimal solution whose leftmost interval starts at x_1
Proof: Suppose there exists an optimal solution whose leftmost interval does not start at x_1. Then its leftmost interval must start before x_1 because otherwise it cannot cover x_1. Assume its first interval covers $[x_0, x_0 + 1]$ where $x_0 < x_1$. If we replace all intervals start earlier than x_1 by $[x_1, x_1 + 1]$, then every point covered by these intervals will still be covered after the shift because x_1 is the very first point and the endpoint before must be smaller than $x_1 + 1$.
If there is only one interval that starts earlier than x_1, then after the replacement, the new solution is also an optimal one.
If there are more than one interval that starts earlier than x_1, then since the new solution contains less intervals, the original solution is not optimal and creates a contradiction. Thus, there is always an optimal solution whose leftmost interval starts at x_1.

Claim 2: There is always an optimal solution where there are no overlapping intervals.
Proof: Suppose there is an optimal solution that contains overlapping intervals. For each group of overlapping intervals, assume that the leftmost interval starts at a and the rightmost interval ends at b. Then this group of intervals can be replaced by a set of non-overlapping unit intervals $S = [[a, a+1], [a+1, a+2]$ ... $[\lceil b\rceil-1, \lceil b\rceil]]$. The number of intervals in S is less than or equal to the number of intervals in the optimal solution. If we do the replacement for each group of overlapping intervals, number of intervals in the final solution is less than or equal to the optimal solution. Thus, the solution obtained from replacement is also optimal.

Combining Claim 1 and 2, we can conclude that our solution is optimal. ■