Problem 1:

Idea: Let $G(V,E)$ be the given complete graph, where $V = \{v_1, v_2, \ldots, v_n\}$. Thus, $n = |V|$. Let $m = n(n-1)/2$, the number of edges in $G$. For each node $v_i$ of the graph, we create a Boolean variable $x_i$. Using the distance value of each edge and the given distances $\alpha_1$ and $\alpha_2$, we create one or two clauses for each edge of $G$. Each clause has exactly two literals. The resulting CNF formula $F = C_1 \land C_2 \land \ldots \land C_r$ (where $r \leq 2m$) is such that there is a partition of $V$ into two sets $V_1$ and $V_2$ with $\text{diameter}(V_1) \leq \alpha_1$ and $\text{diameter}(V_2) \leq \alpha_2$ if and only if there is a satisfying assignment to the 2SAT instance $F$.

(a) High-level description of the algorithm:

1. For each node $v_i$ of $G$, create a Boolean variable $x_i$, $1 \leq i \leq n$. (Intuitively, if $x_i$ is true (false), node $v_i$ will be in subset $V_1$ ($V_2$).)

2. Formula $F$ is initially empty. (Think of $F$ as a set of clauses.)

3. For each edge $e = \{v_i, v_j\}$ of $G$
   
   (a) if $d(v_i, v_j) > \alpha_1$ (Comment: In such a case, $v_i$ and $v_j$ must be in different sets; i.e., $x_i$ and $x_j$ must have opposite truth values.)
   
   Add the two clauses $(\overline{x_i} \lor \overline{x_j})$ and $(x_i \lor x_j)$ to $F$.

   (b) else
   
   if $d(v_i, v_j) > \alpha_2$ (Comment: In such a case, $v_i$ and $v_j$ cannot both be in $V_2$.)
   
   Add the clause $(x_i \lor x_j)$ to $F$.

4. (Note that $F$ is an instance of 2SAT.) If $F$ is satisfiable, then print “Yes” (i.e., there is a partition of $V$ into two sets $V_1$ and $V_2$ satisfying the diameter conditions); otherwise, print “No”.

(b) Proof of correctness:

Theorem: There is a partition of the node set $V$ into two subsets $V_1$ and $V_2$ with $\text{diameter}(V_1) \leq \alpha_1$ and $\text{diameter}(V_2) \leq \alpha_2$ if and only if there is a satisfying assignment to the formula $F$.

Proof:

Part 1: Suppose there is a satisfying assignment to the formula $F$. We will show that there is a partition of $V$ into $V_1$ and $V_2$ satisfying the diameter conditions.

The proof is by contradiction. Consider a satisfying assignment to $F$. For each variable $x_i$ ($1 \leq i \leq n$), if the assignment sets $x_i$ to true, add $v_i$ to $V_1$; otherwise, add $v_i$ to $V_2$. Suppose this partition violates the diameter condition. There are two cases to consider.

Case 1: The diameter of $V_1$ is greater than $\alpha_1$.

Thus, $V_1$ contains two vertices $v_i$ and $v_j$ such that $d(v_i, v_j) > \alpha_1$. Since $v_i$ and $v_j$ are both in $V_1$, the corresponding variables $x_i$ and $x_j$ are both true in the given satisfying assignment. For the edge $\{v_i, v_j\}$, Step 3(a) of the algorithm added the clause $(\overline{x_i} \lor \overline{x_j})$ to the formula $F$. Since both $x_i$ and $x_j$
are true, this clause evaluates to false. This contradicts the assumption that the chosen assignment satisfies all the clauses of $F$.

**Case 2:** The diameter of $V_2$ is greater than $\alpha_2$.

Thus, $V_2$ contains two vertices $v_i$ and $v_j$ such that $d(v_i, v_j) > \alpha_2$. We have the following claim.

**Claim:** $F$ contains the clause $(x_i \lor x_j)$.

**Proof of Claim:** As shown above, $d(v_i, v_j) > \alpha_2$. If $d(v_i, v_j) > \alpha_1$, then for the edge $\{v_i, v_j\}$, Step 3(a) of the algorithm added the clause $(x_i \lor x_j)$ to the formula $F$. If $\alpha_2 < d(v_i, v_j) \leq \alpha_1$, then for the edge $\{v_i, v_j\}$, Step 3(b) of the algorithm added the clause $(x_i \lor x_j)$ to the formula $F$. Thus, in either case, $F$ contains the clause $(x_i \lor x_j)$. This completes the proof of the claim.

We now continue the proof of Case 2. Since $v_i$ and $v_j$ are both in $V_2$, the corresponding variables $x_i$ and $x_j$ are both false in the given satisfying assignment. Since both $x_i$ and $x_j$ are false, the clause $(x_i \lor x_j)$ of $F$ evaluates to false. As in Case 1, this is a contradiction. This completes the proof of Part 1.

**Part 2:** Suppose there is a partition of $V$ into $V_1$ and $V_2$ satisfying the diameter conditions. We will show that there is a satisfying assignment to the formula $F$.

Consider any partition of $V$ into $V_1$ and $V_2$ satisfying the diameter conditions. For $1 \leq i \leq n$, if $v_i \in V_1$, assign the value true to $x_i$; otherwise, assign the value false to $x_i$. We prove by contradiction that the resulting assignment satisfies $F$. So, assume that some clause of $F$ is not satisfied. Note that each clause of $F$ is of the form $(x_i \lor x_j)$ or $(\overline{x_i} \lor \overline{x_j})$. Thus, there are two cases to consider.

**Case 1:** The clause that is not satisfied is of the form $(x_i \lor x_j)$.

Since the clause is not satisfied, both $x_i$ and $x_j$ are false. In other words, nodes $v_i$ and $v_j$ are both in $V_2$. The algorithm added the clause $(x_i \lor x_j)$ because $d(v_i, v_j) > \alpha_1$ or because $\alpha_2 < d(v_i, v_j) < \alpha_1$. Either way, $d(v_i, v_j) > \alpha_2$; that is, diameter($V_2$) > $\alpha_2$. This contradicts the assumption that the chosen partition satisfies the diameter conditions.

**Case 2:** The clause that is not satisfied is of the form $(\overline{x_i} \lor \overline{x_j})$.

Since the clause is not satisfied, both $x_i$ and $x_j$ are true. In other words, nodes $v_i$ and $v_j$ are both in $V_1$. The algorithm added the clause $(\overline{x_i} \lor \overline{x_j})$ because $d(v_i, v_j) > \alpha_1$. However, the chosen partition has both $v_i$ and $v_j$ in $V_1$. Thus, diameter($V_1$) > $\alpha_1$. This contradicts the assumption that the chosen partition satisfies the diameter conditions. This completes the proof of the theorem.

**Problem 2:** For convenience, we will use the following notation in describing the dynamic programming algorithm.

(a) For $1 \leq i \leq n$, let $S_i = s_1 \ldots s_i$ denote the substring of $S$ formed by the first $i$ characters of $S$.

(b) For $1 \leq r \leq i \leq n$, let $T_{ri} = s_r \ldots s_i$ denote the substring of $S$ formed by the characters in positions $r$ through $i$ of $S$.

The dynamic programming table is a one-dimensional array $N[1 .. n]$ with $n$ elements, where $N[i]$ stores the minimum number of pattern strings needed to realize the substring $S_i$, $1 \leq i \leq n$. Once we
compute all the \( n \) values in the array \( N \), the answer to the problem is given by \( N[n] \). (As mentioned in the problem statement, for any value of \( i \), if the substring \( S_i \) is not realizable using \( P \), then we set \( N[i] = \infty \).)

To show how the entries of the array \( N \) can be computed, we will use an auxiliary function \( C \) whose input is a string \( x \). The definition of \( C \) is as follows.

\[
C(x) = \begin{cases} 
  1 & \text{if } x \text{ is a pattern string} \\
  \infty & \text{otherwise.}
\end{cases}
\]

Thus, \( C \) tells us whether or not the string \( x \) is one of the given pattern strings.

To begin with, note that \( N[1] = C(S_1) \). Now, assume that for some \( 2 \leq i \leq n \), we have computed all the values in the subarray \( N[1..i-1] \). The idea for computing \( N[i] \) (which is the minimum number of pattern strings needed to realize the substring \( S_i \)) is the following:

(a) If \( S_i \) is one of the pattern strings, then \( N[i] = 1 \).

(b) Otherwise, in any optimal realization of \( S_i \), the very last substring (which must include the character \( s_i \)) starts with one of \( s_2, s_3, \ldots, s_i \); that is, the last substring is one of \( T_{2i}, T_{3i}, \ldots, T_{ii} \). Now, if the last substring is \( T_{ri} \) (for some \( r \), \( 2 \leq r \leq i \)), then the initial part \( S_{r-1} = s_1 \ldots s_{r-1} \) must also be realized optimally. In other words, the minimum number of pattern substrings needed to realize the substring \( S_{r-1} \) is given by \( N[r-1] \) (which has already been computed).

Therefore, the equation for computing the value of \( N[i] \), for \( 2 \leq i \leq n \), is the following.

\[
N[i] = \min \left\{ C(S_i), \min_{2 \leq r \leq i} \left[ N[r-1] + C(T_{ri}) \right] \right\}. \tag{1}
\]

**High-level Pseudocode:** It is easy to see how the function \( C \) can be implemented for any string \( x \) by doing string comparisons with the \( m \) pattern strings. The pseudocode below assumes the code for computing \( C \) and shows how the entries of array \( N \) can be computed using Equation (1).

1. \( N[1] = C(S_1) \).
2. for \( i = 2 \) to \( n \) do
   
   \[
   N[i] = \min \left\{ C(S_i), \min_{2 \leq r \leq i} \left[ N[r-1] + C(T_{ri}) \right] \right\}.
   \]
3. Output \( N[n] \).

**Running time:** We first consider the time needed to compute the function \( C \). Since each pattern string is of length at most \( k \), the value of \( C \) is \( \infty \) for any string whose length is \( > k \). So, there is no need to do any string comparisons in that case. For any string \( x \) whose length is at most \( k \), we need to do at most \( m \) string comparisons (since the pattern set \( P \) is of size \( m \)) to find the value of \( C(x) \). Each such string comparison can be done in \( O(k) \) time. Thus, for any string \( x \), \( C(x) \) can be computed in \( O(mk) \) time.

Computing the value of \( N[1] \) uses just one call to obtain the value of \( C(S_1) \); thus, the time needed to compute \( N[1] \) is \( O(mk) \). Now consider the computation of \( N[i] \), for any \( i, 2 \leq i \leq n \). Using Equation (1) for \( N[i] \), we need at most \( n \) computations of \( C \) (one for \( S_i \) and one for each substring \( T_{ri}, 2 \leq r \leq i \)). So, the time for all these computations is \( O(nmk) \). Once these values are available, computing the minimum can be done in \( O(n) \) time. Therefore, the time needed to compute \( N[i] \) is
Thus, the time needed to compute all the $n$ entries in $N$ is $O(n^2mk)$. In other words, the running time of the algorithm is $O(n^2mk)$.

---

### Problem 3:

**Basic idea:** Suppose we have an algorithm SUBSEQ which given two strings $P$ and $Q$ of lengths $p$ and $q$ respectively, where $p \geq q$, can determine whether $Q$ is a subsequence of $P$ in $O(p)$ time. (We will see how to design such an algorithm shortly.) Using Algorithm SUBSEQ, we can solve the given problem as follows.

- Since the lengths of strings $A$ and $B$ are $n$ and $m$ respectively, with $m \leq n$, the largest value of $r$ for which $B^r$ can be a subsequence of $A$ is $\lfloor n/m \rfloor$.
- If for some $r$, $B^r$ is not a subsequence of $A$, then for any $t > r$, $B^t$ cannot be a subsequence of $A$. Therefore, we can use binary search in the range $[1 \ldots \lfloor n/m \rfloor]$ to find the largest value of $r$ such that $B^r$ is a subsequence of $A$.
- For any $r$ in the above range, the length of $B^r$ is at most $n$. Therefore, string $B^r$ can be constructed from $B$ in $O(n)$ time. Further, each call to Algorithm SUBSEQ with strings $A$ and $B^r$ uses at most $O(n)$ time.
- Because of the use of binary search, the number of times we construct $B^r$ and call SUBSEQ is at most $\log_2 \lfloor n/m \rfloor = O(\log (n/m))$. The time taken by each construction of $B^r$ and the execution of SUBSEQ is $O(n)$.
- Therefore, using SUBSEQ and binary search, the running time of the algorithm is $O(n \log(n/m))$.

Thus, the only remaining part is to design Algorithm SUBSEQ. Assume that the inputs to Algorithm SUBSEQ are two arrays of characters $P[1 \ldots p]$ and $Q[1 \ldots q]$, and that the algorithm is required to return TRUE if $Q$ is a subsequence of $P$ and FALSE otherwise.

The main idea behind the algorithm is the following. To begin with, it looks for the first occurrence of the character $Q[1]$ in $P$, starting from $P[1]$. If $Q[1]$ can’t be matched, obviously, $Q$ is not a subsequence of $P$ and the algorithm returns FALSE. So, assume that $Q[1]$ was matched with its first occurrence in $P$.

Now, suppose that the algorithm has matched up to $Q[i]$, for some $i \geq 1$, and this character was matched with $P[j]$. Then, the algorithm tries to match the character $Q[i+1]$ with the first occurrence of this character in $P$, starting from $P[j+1]$. This is continued until all characters in $Q$ are matched (when the algorithm returns TRUE) or a failure occurs at some stage (when the algorithm returns FALSE).

We first give the outline of the algorithm. This is followed by the proof of correctness and running time analysis.
Algorithm SUBSEQ($P[1 .. p]$, $Q[1 .. q]$)

1. $j = 1$.

2. for $i = 1$ to $q$ do
   (a) Find the smallest index $k$ in the range $[j .. n]$ such that $Q[i] = P[k]$. If no such index is found, return False.
   (b) $j = k + 1$.

3. return True.

We now prove the correctness of the above algorithm and its running time.

Lemma 1 $Q$ is a subsequence of $P$ if and only if Algorithm SUBSEQ returns True.

Proof: If SUBSEQ returns True, it has matched every character in $Q$ in order with a character in $P$. So, $Q$ is indeed a subsequence of $P$. Now, suppose $Q$ is a subsequence of $P$. We will prove that SUBSEQ returns True.

Since $Q$ is a subsequence of $P$, there exist indices $t_1, t_2, \ldots, t_q \leq p$ such that $1 \leq t_1 < t_2 < \ldots < t_q$ and $Q[i] = P[t_i]$, $1 \leq i \leq q$. To find a character of $P$ that matches $Q[1]$, SUBSEQ starts at index 1. Since $P[t_1] = Q[1]$, SUBSEQ will surely find the first index $w_1 \leq t_1$ such that $P[w_1] = Q[1]$. SUBSEQ starts the search for a character of $P$ that matches $Q[2]$ at index $w_1 + 1 \leq t_1 + 1$. Since $t_1 < t_2$ and $P[t_2] = Q[2]$, SUBSEQ will surely find, starting from $w_1 + 1$, an index $w_2 \leq t_2$ such that $P[w_2] = Q[2]$. Continuing inductively, it is seen that SUBSEQ will surely find an index $w_q \leq t_q$ such that $P[w_q] = Q[q]$. Thus, SUBSEQ returns True, and this completes the proof of Lemma 1. ■

Lemma 2 When $p \geq q$, Algorithm SUBSEQ runs in $O(p)$ time.

Proof: For each character of $Q$ that is matched, Algorithm SUBSEQ examines a segment of $P$. Comparison of a character in $Q$ with a character in $P$ can be done in $O(1)$ time. Thus, the time used to find the match for each character in $Q$ is at most constant times the length of the corresponding segment of $P$. The segments of $P$ examined for matching different characters of $Q$ are pairwise disjoint. Therefore, the total time needed to match all the characters in $Q$ (or to decide that some character of $Q$ can’t be matched, and hence return False) is at most a constant times the length of $P$. In other words, the running time of Algorithm SUBSEQ is $O(p)$. ■