1.

The problem of computing the edit distance of S and T is equivalent to the problem of finding the length of a longest common substring of S and T, where a common substring is a string that equals both a subsequence of S and a subsequence of T. Namely, if S and T of length n have a longest common substring U of length m then their edit distance is $2(n-m)$ (deletions correspond to the difference between S and U, and insertions correspond to the difference between U and T; the minimum-number-of-operation and maximum-string-length conditions are easily seen to be equivalent).

The following dynamic programming algorithm computes the length of a longest common substring.

Given strings $s_1, \dots, s_n$ and $t_1, \dots, t_n$:

1. Let $L[0,\dots,n][0,\dots,n]$ be a two-dimensional array with values in $\{0,\dots,n\}$.
2. for $i = 0,\dots,n$ do
   - for $j = 0,\dots,n$ do
     - if $i > 0, j > 0$ and $s_i = s_j$ then
       - $L[i][j] \leftarrow L[i-1][j-1] + 1$

       else
       - assign to $L[i][j]$ the maximum of the following values:
         $$\begin{array}{ll} 0 & \\ L[i-1][j] & \text{if } i > 1, j > 0 \\ L[i][j-1] & \text{if } i > 0, j > 1 \end{array}$$
3. Output $L[n][n]$

Let $c(\alpha, \beta)$ denote the length of a longest common substring of α, β.

Claim. *For any $i, j \in \{0,\dots,n\}$, the value assigned to $L[i][j]$ is $c(s_1 \cdots s_i, t_1 \cdots t_j)$.*

Proof. By induction on the sequence of iterations. For the base, indeed $L[0][0] = 0$. Consider $L[i][j]$, and suppose the property holds for all previously computed elements of L, and specifically for those mentioned in the update step. Then indeed $L[i][j]$ is assigned the right value:

- If $s_i = t_j$ then $c(s_1 \cdots s_i, t_1 \cdots t_j) = c(s_1 \cdots s_{i-1}, t_1 \cdots t_{j-1}) + 1$: the $\geq$ direction is obvious. For $\leq$, consider the matching of string indices that corresponds to the longest common substring of $s_1 \cdots s_i$ and $t_1 \cdots t_j$. If s_i is matched to some t_k then $t_{k+1}, \dots, t_j$ is unmatched (and similarly when s and t are switched), thus w.l.o.g. we may match s_i to t_j.
- If $s_i \neq t_j$ then at least one of them is unmatched in any common substring of $s_1 \cdots s_i$ and $t_1 \cdots t_j$, so $c(s_1 \cdots s_i, t_1 \cdots t_j) = \max\{c(s_1 \cdots s_{i-1}, t_1 \cdots t_j), c(s_1 \cdots s_i, t_1 \cdots t_{j-1})\}$.

$\square$

The algorithm performs n^2 iterations, each of which consists of a constant number of array accesses and symbol comparisons. Thus in a random-access machine, the time complexity is $O\left(n^2 \log |\Sigma|\right)$. The space complexity is $O(n^2 \log n)$, but can be reduced to $O(n \log n)$ by noting that in iteration i of the outer loop, only $L[i-1][\cdot]$ and $L[i][\cdot]$ are referenced.

2.

The proposition follows from the following two claims:

Claim. *In any graph $G = (V, E)$, for any matching $M \subseteq E$ and any vertex cover $C \subseteq V$, $|M| \leq |C|$.*

Proof. For any edge $(u, v) \in M$ at least one of u and v is in C, for otherwise this edge is not covered. As the edges of M are disjoint, it follows that $|C| \geq |M|$. $\square$

Claim. *In any bipartite graph $G = (V, E)$, there exists a matching $M \subseteq E$ and a vertex cover $C \subseteq V$ such that $|M| \geq |C|$.*

Proof. Consider the alternating forest produced by the algorithm for finding a maximum matching in bipartite graph. In this forest, let O denote the set of odd vertices, let E denote the set of even vertices and let R denote the rest. Let M be the produced matching. The following properties hold:

1. All vertices in E are matched with vertices in O, by the way the forest is built.
2. All vertices in R are matched, by the way the forest is built.
3. $E \cap (O \times R \cup O \times O) = \emptyset$, by the termination condition. Equivalently, $E \subseteq R \times R \cup E \times V$.

From property 1 it follows that $|M \cap (E \times O)| = |E|$. From property 2 it follows that $|M \cap (V \times R)| \geq \frac{1}{2}|R|$. Thus $|M| \geq |E| + \frac{1}{2}|R|$.

Let V_1, V_2 a partition of V s.t. $E \subseteq V_1 \times V_2$. Let A be $R \cap V_1$ or $R \cap V_2$, whichever is smaller (in fact, both must be of the same cardinality). Clearly $|A| \leq \frac{1}{2}|R|$ and A covers all edges in $R \times R$. Consider $C = E \cup A$: by property 3 it is a vertex cover, and $|C| = |E| + \frac{1}{2}|R| \leq |M|$. $\square$

3.

The following algorithm sorts both the nuts and the bolts, essentially by performing two Quicksorts simultaneously:

Algorithm NBSORT$^C(X, Y)$:

Given a set of nuts $X = \{x_1, \ldots, x_n\}$, a set of bolts $Y = \{y_1, \ldots, y_n\}$ and a nut-to-bolt comparison oracle C such that the constraints of the problem are satisfied, output X and Y sorted separately.

1. If $n = 0$ then output two empty lists and halt.
2. Choose $i \in \{1, \ldots, n\}$ uniformly at random.
3. Compare x_i to the elements of Y one by one using C. Let y_j the bolt that matched x_i, let Y_L be the set of bolts that were too small and let Y_H be the set of bolts that were too large.
4. Compare y_j to the elements of $X \setminus \{x_i\}$ one by one using C. Let X_L be the set of nuts that were too small and let X_H be the set of nuts that were too large.
5. Recursively compute:
   $(\tilde{X}_L, \tilde{Y}_L) \leftarrow$NBSORT$^C(X_L, Y_L)$, $(\tilde{X}_H, \tilde{Y}_H) \leftarrow$NBSORT$^C(X_H, Y_H)$.
6. Output $\tilde{X}_L \parallel (x_i) \parallel \tilde{X}_H$ and $\tilde{Y}_L \parallel (y_i) \parallel \tilde{Y}_H$, where "$\parallel$" denotes list concatenation

---

Correctness of algorithm NBSORT can be shown by induction on n, as follows. For $n = 0$ it holds trivially. Suppose NBSORT is correct for all n' s.t. $n' < n$, and consider its operation on inputs of size n. Let $\chi_1, \ldots, \chi_n$ denote the elements of X ordered by increasing size, and let $\psi_1, \ldots, \psi_n$ denote the elements of Y ordered by increasing size. Then in the top level of NBSORT, letting k be such that $x_i = \chi_k$, the following clearly holds: $y_j = \psi_k$, $X_L = \{\chi_1, \ldots, \chi_{k-1}\}$, $X_H = \{\chi_{k+1}, \ldots, \chi_n\}$, $Y_L = \{\psi_1, \ldots, \psi_{k-1}\}$, $Y_H = \{\psi_{k+1}, \ldots, \psi_n\}$. Thus correctness follows from the induction hypothesis.

The analysis of comparison complexity similar to that of Quicksort. Consider an invocation of NBSORT (including all recursive calls), and let χ_s and ψ_t be defined as above. Let the random variable A_{st} be the indicator variable of the event "χ_s was compared to ψ_t". Let the random variable A be total the number of comparisons performed. Clearly no pair of elements is compared more than once, so $A = \sum_{s,t=1}^{n} A_{st}$.

To evaluate $\mathbb{E}A_{st} = \Pr[A_{st} = 1]$, observe how the random choices of step 2 affect it. Suppose $s < t$ (the other case is similar). Only choices of i for which $x_i = \chi_k$ and $k \in \{s, \ldots, t\}$ are relevant (otherwise $\chi_s, \ldots, \chi_t$ and $\psi_s, \ldots, \psi_t$ are passed to the same recursive invocation but none of them is compared immediately). For $k \in \{s, \ldots, t\}$, χ_s and ψ_t are compared iff $k = s$ or $k = t$ (this holds even for $s = t$, unlike the case of Quicksort). Thus $\forall s : \mathbb{E}A_{ss} = 1$, $\forall s \neq t : \mathbb{E}A_{st} = \frac{|\{s,t\}|}{|\{s,\ldots,t\}|} = \frac{2}{|s-t|+1}$, and by linearity of expectation:

$$
\begin{aligned}
\mathbb{E}A = \sum_{s,t=1}^{n} \mathbb{E}A_{st} &= \sum_{1\le s\le n} \mathbb{E}A_{ss} + \sum_{1\le s<t\le n} \mathbb{E}A_{st} + \sum_{1\le s<t\le n} \mathbb{E}A_{ts} \\
&= n + 2 \sum_{1\le s<t\le n} \frac{2}{s-t+1} = n + 4\sum_{s=1}^{n}\sum_{t=s+1}^{n} \frac{1}{s-t+1} \\
&= n + 4\sum_{s=1}^{n}\sum_{t=2}^{n-s+1} \frac{1}{t} \\
&< n + 4\sum_{s=1}^{n}\int_{1}^{n-s+1} \frac{1}{x}dx = n + 4\sum_{s=1}^{n} \ln(n-s+1) = n + 4\sum_{s=1}^{n} \ln s \\
&< n + 4\int_{1}^{n+1} \ln x\, dx = n + 2(n+1)\ln(n+1) \\
&< (4+\varepsilon)\, n\ln n = O(n\ln n) \qquad \text{(for any } \varepsilon > 0 \text{ and sufficiently large } n\text{)}
\end{aligned}
$$

Asymptotically, in the first two inequalities the ratio between the two sides approaches 1. Hence the multiplicative constant hidden in the O notation asymptotically approaches 4.

The nut-to-bolt matching is easily obtained from the sorted lists, by taking the pairs with corresponding indices.

Remark: The randomized algorithm presented above is not optimal in terms of the expected number of comparison. It is easy to improve the low order terms in the running time by paying more attention to details. For example, when $n = 2$ one comparison suffices in order to match nuts to bolts. An improved leading constant in the expected running time can be obtained by more major changes to the algorithm.

4.

The following is an algorithm for finding a 9-cliques if one exists.

Given $G = (V, E)$, $|V| = n$:

1. Compute the adjacency matrix A of the following graph $G' = (V', E')$:
   - $V' = \{\text{3-cliques in } G\}$
   - $E' = \{\text{pairs of disjoint 3-cliques whose union is a 6-clique}\} = \{(S, T) :\ S, T \in V',\ S \cap T = \emptyset,\ S \times T \subseteq E\}$

   This computation is done using simple enumeration.
2. Compute A^3 using fast matrix multiplication and check if there are nonzero elements on the diagonal of A^3. If none were found, answer "no 9-clique".
3. Let S be a vertex of G' such that in A^3, the diagonal entry corresponding to S is nonzero. Use exhaustive search to select a set R of six vertices in G such that $S \cup R$ form a 9-clique in G. Output $S \cup R$.

Correctness follows from these two observations:

1. A set of vertices is a 9-clique iff it can be partitioned into 3 disjoint 3-cliques such that the union of any two of them forms a 6-clique.
2. For any adjacency matrix A, nonzero entries on the diagonal of A^3 correspond to vertices that reside on cycles of length 3, i.e., to vertices that are members of a 3-clique.

Let ω the exponent of the best known fast matrix multiplication algorithm ($2 < \omega < 3$, and specifically $\omega < 2.376$). In a random-access machine, the computation of V' takes time $O(n^3)$, and $|V| < n^3$. The computation of E' takes time $O(|V'|^2) = O(n^6)$. Step 2 multiplies matrices of size $|V'| \times |V'|$, so it takes time $O(|V'|^\omega) = O\left(n^{3\omega}\right)$. Step 3 takes time $O(n^6)$. Thus the overall time complexity is $O\left(n^{3\omega}\right)$.