Problem 1:

High-level description of the algorithm:

1. Sort the integers in $A$ into non-increasing order. Without loss of generality, let $a_1 \geq a_2 \geq \ldots \geq a_n$ denote this sorted order.

2. Sort the integers in $B$ into non-increasing order. Without loss of generality, let $b_1 \geq b_2 \geq \ldots \geq b_n$ denote this sorted order.

3. The payoff value $p$ is given by
   \[ p = \prod_{i=1}^{n} a_i b_i. \]  

Proof of correctness: The following theorem establishes the correctness of the algorithm.

**Theorem 1.1:** The above algorithm produces the maximum payoff value.

**Proof:** Consider any optimal solution and let $p^*$ denote its payoff value. We will prove that the payoff value $p$ given by Equation (1) has the property that $p = p^*$.

As mentioned in the algorithm, we will assume that $a_1 \geq a_2 \geq \ldots \geq a_n$ and that $b_1 \geq b_2 \geq \ldots \geq b_n$. Since any solution must use each of the integers in $A$ as the base in computing the payoff, we can assume that $p^*$ is given by
   \[ p^* = \prod_{i=1}^{n} a_i^{b_i}, \]  

where the sequence $(r_1, r_2, \ldots, r_n)$ of integers is a permutation of $\{1, 2, \ldots, n\}$. We now show that the chosen optimal solution can be modified, without decreasing the payoff, so that it has the same form as the payoff value $p$ given by Equation (1). This will imply that $p = p^*$ and complete the proof of the theorem.

Let $k$ be the largest integer such that the expressions for $p^*$ and $p$ agree in the first $k$ terms. (The value of $k$ may be 0 since the expressions for $p$ and $p^*$ may differ in the first term itself.) If $k = n$, there is nothing to prove (since the expressions for $p$ and $p^*$ agree on all the terms). Further, $k \neq n - 1$ since if the expressions for $p$ and $p^*$ agree on the first $n - 1$ terms, then they must also agree on the $n$th term. So, $0 \leq k \leq n - 2$, and the expressions for $p$ and $p^*$ can be written in the following form:

\[ p = a_1^{b_1} a_2^{b_2} \ldots a_k^{b_k} \left( a_{k+1}^{b_{k+1}} \ldots a_n^{b_n} \right) \]

\[ p^* = a_1^{b_1} a_2^{b_2} \ldots a_k^{b_k} \left( a_{k+1}^{b_{k+1}} \ldots a_n^{b_n} \right) \]

We now show that $p$ and $p^*$ can be made to agree on the first $(k+1)$ terms.

In the above expressions for $p$ and $p^*$ (Equation (3)), since $k$ is the largest value such that $p$ and $p^*$ agree on the first $k$ terms, we must have $r_{k+1} \neq k + 1$. Let the term $a_j$ (where $j > k + 1$) use
the exponent $b_{k+1}$. Consider the value $p''$ obtained by switching the exponents of $a_{k+1}$ and $a_j$ in the expression for $p^*$ while leaving the other terms unchanged:

$$
p'' = a_1^{b_1} a_2^{b_2} \cdots a_k^{b_k} \left( a_{k+1}^{b_{k+1}} \cdots a_j^{b_j} \cdots a_n^{b_n} \right)
$$

From Equations (3) and (4), we have

$$
p''/p^* = \left( \frac{a_{k+1}^{b_{k+1}} a_j^{b_j}}{a_k^{b_k} a_j^{b_j}} \right) (b_{k+1} - b_j) \frac{a_{k+1}}{a_j}
$$

Since $a_{k+1} \geq a_j$ and $b_{k+1} \geq b_j$, it follows that $p''/p^* \geq 1$ or $p'' \geq p^*$. Thus, we can modify any optimal solution so that the first $(k+1)$ terms of the payoff expression have the form $a_1^{b_1} \cdots a_{k+1}^{b_{k+1}}$; in other words, the expressions for $p$ and $p^*$ agree up to the first $k+1$ terms.

By repeating the above process, it can be seen that $p$ and $p^*$ can be made to agree in all the terms; that is, $p$ is the optimal payoff value.

**Running time:** The algorithm needs to output just the orderings of the elements of $A$ and $B$. Since this only requires us to sort $A$ and $B$, the algorithm can be implemented to run in $O(n \log n)$ time using (say) Merge-Sort.

---

**Problem 2:**

**Definition:** A unit interval $u = [p, p+1]$ covers a point $x$ if $p \leq x \leq p+1$.

**Idea behind the algorithm:** Start the first unit interval from the left most point in the input (greedy choice) and delete points covered by this interval. Repeat these steps on the remaining set of points until all points are covered.

**High-level description of the algorithm:**

1. Sort the given set of points into nondecreasing order. Let $(x_1, x_2, \ldots, x_n)$ denote the sorted order.
   Initialize the solution set $S$ to empty.

2. Let $C_1 = [x_1, x_1 + 1]$. Add the (closed unit) interval $C_1$ to $S$.
   Set $\text{RightEnd} = x_1 + 1$ and $j = 1$. (Thus, all the points up to and including $\text{RightEnd}$ have already been covered by the intervals in $S$.)

3. for $i = 2$ to $n$ do
   if $(x_i > \text{RightEnd})$ // Point $x_i$ has not yet been covered.
     (a) $j = j + 1$.
     (b) Let $C_j = [x_i, x_i + 1]$. Add the interval $C_j$ to $S$.
     (c) Set $\text{RightEnd} = x_i + 1$.

4. Output the solution set $S$.  


**Proof of correctness:** We begin with some notation. Let \( X = \langle x_1, x_2, \ldots, x_n \rangle \) be the given set of points in non-decreasing order. Let \( S = \langle C_1, C_2, \ldots, C_t \rangle \) denote the set of \( t \) unit intervals chosen by the algorithm in that order. Let \( S^* = \{D_1, D_2, \ldots, D_{t^*}\} \) denote an optimal set of \( t^* \) unit intervals that cover the points of \( X \), where again the intervals are in sorted order of their left end points.

If two of the intervals in \( S^* \) have the same left end point, then they would be identical unit intervals; thus, one of them can be deleted and this would violate the optimality of \( S^* \). Therefore, we assume throughout this discussion that the left end points of the intervals in \( S^* \) are strictly in increasing order.

For any interval \( I \), let \( L(I) \) and \( R(I) \) denote respectively the left and right end points of \( I \). We begin with an observation which follows directly from the description of the algorithm.

**Observation 2.1:** For each unit interval \( C_j \) in \( S \), \( L(C_j) \) is a point in \( X \). □

The following lemma formally captures the intuition that “greedy does not fall behind optimum”.

**Lemma 2.2:** For \( 1 \leq j \leq t^* \), \( L(C_j) \geq L(D_j) \).

**Proof:** The proof is by induction on \( j \).

- **Basis:** For \( j = 1 \). Note that \( L(C_1) = x_1 \). If \( L(D_1) > x_1 \), then \( S^* \) cannot cover \( X \). Therefore, \( L(C_1) \geq L(D_1) \), and the basis holds.

- **Inductive hypothesis:** Assume that the result is true for \( j = r \), for some \( r \geq 1 \).

- **To prove:** The result is also true for \( j = r + 1 \); that is, we must show that \( L(C_{r+1}) \geq L(D_{r+1}) \). This can be done as follows.

  By Observation 2.1, \( L(C_{r+1}) \) is some point \( x_i \) of \( X \). By the inductive hypothesis, \( L(C_r) \geq L(D_r) \). Since each interval is of length 1, it follows that the \( R(C_r) \geq R(D_r) \). Thus, intervals \( D_1, D_2, \ldots, D_r \) in \( S^* \) don’t cover \( x_i \). Now, if \( L(D_{r+1}) > L(C_{r+1}) \), then point \( x_i \) can’t be covered by \( S^* \), a contradiction. Thus, \( L(C_{r+1}) \geq L(D_{r+1}) \), and this completes the inductive proof. □

The following theorem shows that the algorithm produces an optimal solution.

**Theorem 2.3:** The greedy algorithm uses the minimum number of unit intervals to cover all the points in \( X \).

**Proof:** The proof is by contradiction. Suppose the algorithm is not optimal; that is, \( t > t^* \). Consider the interval \( t^* \). By Lemma 2.2, \( L(C_{t^*}) \geq L(D_{t^*}) \). Since each interval is of length 1, we also have \( R(C_{t^*}) \geq R(D_{t^*}) \).

After choosing the interval \( C_{t^*} \), the greedy algorithm chose one more interval since there was some point \( x_k \in X \) which has not been covered by the first \( t^* \) intervals in \( S \). In other words, \( x_k > R(C_{t^*}) \). Since \( R(C_{t^*}) \geq R(D_{t^*}) \), it follows that \( x_k > R(D_{t^*}) \). Thus, point \( x_k \) is not covered by \( S^* \), a contradiction. The theorem follows. □

**Running time:** Step 1 (sorting) can be done in \( O(n \log n) \) time. Step 2 takes \( O(1) \) time. Step 3 takes \( O(n) \) time since we examine \( O(n) \) points and the body of the loop takes only \( O(1) \) time per point. So, the overall running time of the algorithm is \( O(n \log n) \).
Problem 3:

For each node \( i \), \( 1 \leq i \leq n \), our dynamic programming algorithm maintains two values, namely by \( X[i] \) and \( Y[i] \), defined as follows.

(a) \( X[i] \) denotes the size of a maximum independent set for the subtree rooted at \( i \) under the condition that node \( i \) is not included in the maximum independent set.

(b) \( Y[i] \) denotes the size of a maximum independent set for the subtree rooted at \( i \) under the condition that node \( i \) is included in the maximum independent set.

The computation steps are as follows.

1. For a leaf node \( i \), the subtree rooted at \( i \) consists of just that node. Therefore, \( X[i] = 0 \) and \( Y[i] = 1 \). Clearly, in this case, the time need to compute \( X[i] \) and \( Y[i] \) is \( O(1) \).

2. Given that \( i \) is an internal node whose left and right children are \( j \) and \( k \), we can compute \( X[i] \) and \( Y[i] \) as follows.

   (i) In computing \( X[i] \), we note that \( i \) must be excluded from the independent set. Therefore, we are free to choose or not choose \( j \) and \( k \) in the independent set. Therefore,

   \[ X[i] = \max\{X[j], Y[j]\} + \max\{X[k], Y[k]\}. \]

   (ii) In computing \( Y[i] \), we note that \( i \) must be included in the independent set. Therefore, we cannot choose either \( j \) or \( k \) in the independent set. Therefore,

   \[ Y[i] = 1 + X[j] + X[k]. \]

   Clearly, in this case also, \( X[i] \) and \( Y[i] \) can be computed in \( O(1) \) time.

   Thus, each entry of \( X \) and \( Y \) can be computed in \( O(1) \) time. Since the total number of entries to be computed is \( 2n \), the time needed to compute all the entries of \( X \) and \( Y \) is \( O(n) \).

3. Note that the root is numbered \( n \). Therefore, after computing all the entries of \( X \) and \( Y \), the size of a maximum independent set can be computed as \( \max\{X[n], Y[n]\} \). Clearly, the time needed for the computation is \( O(1) \).

Thus, the overall running time of the algorithm is \( O(n) \).