Problem 1: Let \( C[1..2n] \) denote the array that contains the result of merging \( A \) and \( B \). In the array \( C \), the elements of \( A \) must appear in the same order as in \( A \); likewise, the elements of \( B \) must also appear in the same order as in \( B \).

Each way of merging \( A \) and \( B \) can be described by choosing \( n \) positions in \( C \), filling the chosen positions using the elements \( A[1] \) through \( A[n] \) in that order from left to right, and filling the remaining \( n \) positions with the elements \( B[1] \) through \( B[n] \) in that order from left to right. Thus, the number of possible ways of merging \( A \) and \( B \) is equal to the number of ways of choosing \( n \) positions from a set of \( 2n \) positions. In other words, the number of possible ways to merge \( A \) and \( B \) is \( \binom{2n}{n} \).

Part (b): Consider a decision tree representing a comparison-based algorithm that merges two sorted arrays of size \( n \). From the result of Part (a), the number of leaves in the decision tree must be \( \binom{2n}{n} \). Thus, the height of the tree, that is the number of comparisons in the worst-case, must be at least \( \log_2 \binom{2n}{n} \). To show that this quantity is at least \( 2n - o(n) \), we use Stirling’s approximation:

**Fact 3:** For any positive integer \( k \), 
\[
\sqrt{2\pi k} \left(\frac{k}{e}\right)^k \leq k! \leq \sqrt{2\pi k} \left(\frac{k}{e}\right)^{k+1}.
\]

Note that \( \binom{2n}{n} = \frac{(2n)!}{(n!n!)} \). Using Fact 3, \( \frac{(2n)!}{(n!n!)} \geq 2^{2n}e^2/(\sqrt{\pi n^{2.5}}) \). Therefore,
\[
\log_2 \left[ \frac{(2n)!}{(n!n!)} \right] \geq 2n - (2.5 \log_2 n + \log_2 \sqrt{\pi} - \log_2 e^2).
\\\n(1)
\]
Since \( \log_2 n = o(n) \), the right hand side of Inequality (1) is \( 2n - o(n) \). This completes the proof.

Problem 2:

**Idea:** We first use \( A \) to find the lower median \( x \) of \( S \). If \( i \) specifies the position of the lower median of \( S \), then the \( i^{th} \) smallest is indeed \( x \). Otherwise, we use the \textsc{Partition} function on \( S \) to get low and high sides of \( S \). We recurse on the appropriate side of \( S \), depending upon the relationship between \( i \) and the position of the median of \( S \). Since we use the median element as the pivot in \textsc{Partition}, larger of the two sides of \( S \) has at most \( \lceil n/2 \rceil \) elements. This leads to the running of \( O(n) \).

**Algorithm:** Since the algorithm is recursive, we use the parameters \( a \) and \( b \) to specify a subarray of \( S \). The parameter \( r \) indicates that we want to find the \( r^{th} \) smallest value in \( S[a..b] \). (The initial call to the algorithm is \textsc{Order-Statistic} \((S, 1, n, i)\).)
Order-Statistic $(S, a, b, r) \ /*$ Returns the $r$th smallest in $S[a..b]$. */

1. Let $p = b-a+1$ and let $m = \text{floor}((p+1)/2)$. Use the given
   algorithm $A$ to find the lower median $x$ of $S[a..b]$.
   (Thus, $A$ returns the $m$th smallest value in $S[a..b]$.)
2. if $(r = m)$ then return $x$
   else
   2.1. Use Partition on $S[a..b]$ using $x$ as the pivot.
   Let $q$ be the index returned by Partition.
   2.2 if $(r < m)$ then /* Find the $r$th smallest on the low side of $S$. */
       Order-Statistic $(S, a, q-1, r)$
   else /* Find the $(r-m)$th smallest on the high side of $S$. */
       Order-Statistic $(S, q+1, b, r-m)$

**Correctness:** Suppose $|S| = n$ and we are looking for the $i$th smallest value. Algorithm $A$ returns the lower median $x$, that is, the $m$th smallest value, where $m = \lfloor (n+1)/2 \rfloor$. The correctness of the algorithm is a consequence of the following observations:

(a) If $i = m$, then the required value is indeed $x$.

(b) If $i < m$, then the $i$th smallest remains the the $i$th smallest on the low side of the partition.

(c) If $i > m$, then the $i$th smallest is the $(i-m)$th smallest on the high side of the partition.

**Running time analysis:** Let $T(n)$ be the running time of the algorithm on an array of size $n$. The call to Algorithm $A$ uses $O(n)$ time. The call to PARTITION also runs in $O(n)$ time. Since the median value is used as the pivot, the subsequent recursive call is on a subarray of size at most $\lceil n/2 \rceil$. The recursion ends when the subarray size is 1. Therefore, the recurrence for $T(n)$ is:

$$T(n) \leq T(\lceil n/2 \rceil) + cn$$

for $n \geq 2$ and $T(1) = c_1$, for some constants $c$ and $c_1$.

We can solve the above recurrence using the Master Theorem. Comparing with the Master Theorem template, we note that $a = 1$, $b = 2$ and $f(n) = cn$. Thus, $\log_b a = 0$ and $n^{\log_b a} = n^0 = 1$. Thus, $f(n) = \Omega(n^{\log_b a + \epsilon})$, with $\epsilon = 1$. Thus, if the regularity condition holds, then Part 3 of Master Theorem can be applied.

The regularity condition holds since $af(n/b) = cn/2 = (1/2)c n$. Therefore, by Part 3 of Master Theorem, $T(n) = \Theta(f(n)) = \Theta(n)$. Thus, the running time is indeed linear in $n$. 


**Problem 3:** For \( k = 1 \), the value \( x_1 \) is stored with probability 1. Thus, the result holds for \( k = 1 \) trivially. So, we assume that \( k \geq 2 \).

Consider any \( x_i \), where \( 1 \leq i \leq k \). At the end of the \( k^{th} \) step, the value stored will be \( x_i \) if and only if both of the following conditions hold:

1. At Step \( i \), the value \( x_i \) replaced the stored value.
2. At each of the subsequent steps \( i + 1, i + 2, \ldots, k \), the stored value was *not* replaced.

Letting \( E_1 \) and \( E_2 \) denote the two events above, we see that the required probability is given by

\[
\Pr\{E_1\} \times \Pr\{E_2\}.
\]

Note that \( \Pr\{E_1\} = 1/i \) since in Step \( i \), the value \( x_i \) replaces the stored value with probability \( 1/i \). For each subsequent step \( j \) \( (i + 1 \leq j \leq k) \), the probability that \( x_i \) is *not* replaced is given by

\[
1 - \left(\frac{1}{j}\right) = \frac{j - 1}{j}.
\]

Therefore, the required probability = \( \Pr\{E_1\} \times \Pr\{E_2\} = 1/k \).

---

**Problem 4:**

**Notation:** For each node \( v_i \), \( p_i \) denotes the profit obtained by placing a restaurant at \( v_i \). We use \( d(v_i, v_j) \) to denote the distance between nodes \( v_i \) and \( v_j \). Recall that every pair of restaurants must be separated by a distance of at least \( k \).

**Main idea:** For each node \( v_i \), an optimal solution may or may not place a restaurant at that node. The dynamic programming algorithm keeps track of both of these possibilities.

For each node \( v_i \), we compute and store two values, denoted by \( A[i] \) and \( B[i] \), \( 1 \leq i \leq n \). Here, \( A[i] \) \( (B[i]) \) represents the maximum profit for the subproblem \( \langle v_1, \ldots, v_i \rangle \) when a restaurant is placed (not placed) at \( v_i \). After computing all the \( n \) entries of \( A \) and \( B \), the solution to the problem is given by \( \max\{A[n], B[n]\} \). We now discuss how the values in arrays \( A \) and \( B \) can be computed.

By definition, \( A[1] = p_1 \) and \( B[1] = 0 \). Now, suppose we have calculated values \( A[1 \ldots i] \) and \( B[1 \ldots i] \) for some \( i \geq 1 \). We can compute \( A[i + 1] \) and \( B[i + 1] \) as follows.
(a) $B[i + 1] = \max\{A[i], B[i]\}$. (Reason: If a restaurant is not placed at $v_{i+1}$, then the best profit for the subproblem $⟨v_1 \ldots v_{i+1}\rangle$ is the same as that for the subproblem $⟨v_1 \ldots v_i⟩$.)

(b) Try to find the first node $v_j$ to the left of $v_{i+1}$ (i.e., largest $j \in [1..i]$) such that $d(v_j, v_{i+1}) \geq k$. If such a node $v_j$ is found, then $A[i + 1] = p_{i+1} + \max\{A[j], B[j]\}$; otherwise, $A[i + 1] = p_{i+1}$. (Reason: If a restaurant is placed at $v_{i+1}$, then restaurants can’t be placed at any of the nodes that are at a distance less than $k$ from $v_{i+1}$.)

Pseudocode for the algorithm:


2. for $i = 1$ to $n - 1$ do
   
   (a) $B[i + 1] = \max\{A[i], B[i]\}$.

   (b) Try to find the first node $v_j$ to the left of $v_{i+1}$ such that $d(v_j, v_{i+1}) \geq k$. If such a node $v_j$ is found, then $A[i + 1] = p_{i+1} + \max\{A[j], B[j]\}$; otherwise, $A[i + 1] = p_{i+1}$.

3. Output $\max\{A[n], B[n]\}$ as the solution.

Running time analysis: Steps 1 and 3 take $O(1)$ time. So, the dominant part of the running time is due to Step 2.

To implement Step 2 efficiently, we first compute the distance $D[i]$ of each node $v_i$ from $v_1$, $1 \leq i \leq n$. (In other words, $D[i] = d(v_1, v_i)$, $1 \leq i \leq n$.) To do this, note that $D[1] = 0$ and $D[i] = D[i - 1] + d(v_{i-1}, v_i)$, $2 \leq i \leq n$. Since the $d(v_i, v_{i-1})$ values are given, we can compute all the $n$ entries of $D$ in $O(n)$ time. Once we have all the entries of $D$, notice that for any $v_i$ and $v_j$, where $j < i$, the value $d(v_j, v_i) = D[i] - D[j]$ can be computed in $O(1)$ time.

For each iteration of the loop in Step 2, Step 2(a) runs in $O(1)$ time. In Step 2(b), using the array $D$, we can determine whether node $v_j$ exists, and if so, find the node in $O(n)$ time. (This uses a simple backward scan from $v_{i+1}$.) After this, it takes only $O(1)$ time to compute the value of $A[i + 1]$. Thus, Step 2(b) runs in $O(n)$ time. Since the for loop runs $n - 1$ times, the time for Step 2 and hence the running time of the algorithm is $O(n^2)$.

Exercise to the student: It is possible to implement the above algorithm so that its running time is $O(n \log n)$. Try to develop such an implementation.