1. Here is a Las Vegas algorithm:

\[ \text{repeat} \]
\[ \quad 1) \text{Pick a random sample } S \text{ from } A[1:n] \text{ of size } 8\alpha \sqrt{n} \log n; \]
\[ \quad 2) \text{Sort the sample } S, \text{ scan through it to see if there are multiple copies} \]
\[ \quad \text{of any element. If so, output this element and quit;} \]
\[ \text{forever} \]

**Analysis:** Let \( X \) be the number of copies of the repeated element in the sample. Clearly, \( X \) is binomially distributed with parameters \( 8\alpha \sqrt{n} \log n \) and \( \frac{\sqrt{n}}{n} \). The mean of \( X \) is \( 8\alpha \log n \).

Using Chernoff bounds, \( \Pr[X < (1-\epsilon)8\alpha \log n] \leq \exp \left( -\frac{\epsilon^2 8\alpha \log n}{2} \right) \). Picking \( \epsilon = 1/2 \), \( \Pr[X < 4\alpha \log n] \leq n^{-a} \). Thus it follows that the \textbf{repeat} loop is executed only once with high probability.

Step 1 takes \( O(\sqrt{n} \log n) \) time. Sorting in step 2 takes \( O(\sqrt{n} \log^2 n) \) time. Looking for the repeated element in the sorted sample takes \( O(\sqrt{n} \log n) \) time. Put together, the run time of the algorithm is \( \tilde{O}(\sqrt{n} \log^2 n) \) time.

2. Here is a possible algorithm:

Construct \( n \) singleton sets \{1\}, \{2\}, \ldots, \{n\};
/* Every node of the graph corresponds to a set */
Let \( E := \{e_1, e_2, \ldots, e_m\} \);
for \( i := 1 \) to \( m \) do
  Let \( e_i = (a, b) \);
  \( c := \text{CollapsingFind}(a); \ d := \text{CollapsingFind}(b); \)
  if \( c \neq d \) then \text{WeightedUnion}(c, d);
/* At this point, if there is a path between any two nodes } u \text{ and } v \text{ then they will be in the same set */
for \( i := 1 \) to \( n \) do
  \( c := \text{CollapsingFind}(u_i); \ d := \text{CollapsingFind}(v_i); \)
  if \( c = d \) then \text{output} “There is a path between } u_i \text{ and } v_i”; 
else \text{output} “There is no path between } u_i \text{ and } v_i”;

In the above algorithm there are \( 2m + 2n \) \text{Find operations and at most } n \text{ Union operations. Therefore, the run time of the algorithm is } O((m + n)\alpha(m + n)) \text{ where } \alpha \text{ is the inverse Ackermann’s function.}
3. (a) Here \( a = 27, b = 3, \) and \( f(n) = n^4. \) \( n^{\log_b a} = n^3. \) Also, \( a f(n/b) = 27 \frac{n^4}{81} = \frac{n^4}{3} \) and hence the regularity condition holds. As a result, case 3 of Master theorem holds. Therefore, \( T(n) = \Theta(n^4). \)

(b) We can use repeated substitutions here. \( T(n) = T(n-1) + \log_e n = T(n-2) + \log_e (n-1) + \log_e n = T(n-3) + \log_e (n-2) + \log_e (n-1) + \log_e n. \) Continuing this, it follows that \( T(n) = 1 + \sum_{i=3}^{n} \log_e i. \) We can approximate this summation with the integral \( \int_3^n \log_e i \, di = (i \log_e i - i)^n_3 = \Theta(n \log n). \) Therefore, \( T(n) = \Theta(n \log n). \)

4. Informal proof of correctness: Call the first third, the second third, and the third third of the array \( B, C, \) and \( D, \) respectively. Let the largest one third, the next largest one third, and the smallest one third of the array elements be called \( A_1, A_2, \) and \( A_3, \) respectively. When we sort the first two thirds of the array, all the elements of \( A_1 \) in regions \( B \) and \( C \) will move to region \( C. \) When we sort the last two thirds, these elements will move to region \( D \) (where they belong to in sorted order). Also, all the elements of \( A_1 \) that were in region \( D \) will not have moved out. Therefore, after the second sorting step, all the elements of \( A_1 \) would have moved to region \( D \) and also they would get sorted.

Similarly, the second and third sorting steps ensure that all the elements of \( A_3 \) end up in region \( B \) and they get sorted. Put together, the elements of \( A_2 \) end up in region \( C \) and they also get sorted in the third sorting step.

Run time analysis: The run time, \( T(n), \) of this algorithm satisfies: \( T(n) = 3T \left( \frac{2}{3} n \right) + \Theta(1). \) Here \( a = 3, b = 3/2, \) and \( f(n) = \Theta(1). \) \( n^{\log_b a} = n^{\log_{3/2} 3} = O(n^{2.71}). \) Clearly, case 1 of Master theorem holds here. Thus, \( T(n) = O(n^{2.71}). \)

5. Recall that to prove a lower bound on the number of comparisons needed to sort \( n \) elements, we observed that there are \( n! \) possible answers and hence any comparison tree to sort \( n \) elements will have at least \( n! \) leaves. As a result, the height of any such comparison tree will have to be \( \geq \log(n!). \) Therefore, \( \log(n!) \) is a lower bound on the number of comparisons in the worst case.

For the problem of merging also, we can use a similar technique. Let \( A \) and \( B \) be two sorted sequences that we are interested in merging (with \( |A| = |B| = n). \) When we merge \( A \) and \( B \) how many possible answers are there? The answer is \( \binom{2n}{n}. \) One way of seeing this is as follows: Consider a grid with \( 2n \) points. We want to place the elements of \( A \) and \( B \) in this grid with one element per grid point. Once we choose the \( n \) grid points for placing the elements of \( A, \) the location of each element of \( B \) is automatically fixed. There are \( \binom{2n}{n} \) ways of choosing the locations for the elements of \( A. \)

Therefore, a lower bound on the number of comparisons needed to merge \( A \) and \( B \) is \( \log \binom{2n}{n}. \) Stirling’s approximation for \( k! \) is \( \sqrt{2\pi k} (k/e)^k. \) Using this approximation, \( \binom{2n}{n} = \frac{(2n)!}{(n)!^2} \approx \sqrt{4\pi n} \left( \frac{2n}{e} \right)^{2n} \frac{1}{2^n n^{2n} e^{2n}} = 2^{2n} \frac{n^{2n}}{\sqrt{\pi n}}. \) Thus, \( \log \binom{2n}{n} \approx 2n - \log \left( \frac{\pi n}{2} \right). \)
Put together, $2n - (1/2) \log(\pi n)$ is a lower bound on the number of comparisons needed to merge two sorted sequences of length $n$ each.

6. Create an array $A'$ by replacing every $a[i] \in A$, by $x - a[i]$. Put the elements of $A'$ and $B$ into an array $C$ and create an additional field for every element of $C$ indicating whether this element came from the array $A'$ or $B$. Sort $C$ using radix sort that takes $O(n)$ time. Scan $C$, looking for two adjacent elements that have the same value but were originally in different arrays. Presence of two such keys would indicate that there are two elements $a[i] (\in A)$ and $b[j] (\in B)$ such that $x - a[i] = b[j] \Rightarrow x = a[i] + b[j]$. If no such keys are found, output “NO” and exit. All the operations take $O(n)$ time and space.