Dr. Amotz Bar-Noy’s
Compendium of Algorithms Problems

Problems, Hints, and Solutions
Chapter 1

Searching and Sorting Problems
1.1 Array with One Missing

1.1.1 Problem

Let $A[1], \ldots, A[n]$ be an array of $n$ distinct integers, in which each integer is in the range $[1..n + 1]$. That is, exactly one integer out of $\{1, \ldots, n + 1\}$ is missing from $A$. The goal is to find the missing number as efficiently (fast) as possible.

For each of the following scenarios, describe an efficient algorithm to find the missing integer, justify the correctness of the algorithm, and analyze its time complexity in the worst case.

1. The integers in $A$ appear in an arbitrary order.
1.1.2 Hints

1. Consider the subsequence of the natural numbers, [1, M]. Note that the number of bits it takes to represent their sum is only one more than the number of bits it takes to represent M.

2. Suppose that A has 1024 elements and that A[512] = 511.
1.1.3 Solution

1. Since all the integers from 1 to \((n + 1)\) except \(n\) appear in the array once, simply sum them up as they are scanned, and subtract that total from the sum of all integers between 1 and \(n + 1\) (which appears on the left-hand side of the expression in the formula below). Since the array is only scanned a single time (requiring one addition each step) this algorithm is linear \(O(n)\):

\[
missing = \frac{n^2 + 3n + 2}{2} - \sum_{i=1}^{n} A[i]
\]

2. Use a binary-search approach to find the smallest element that is not equal to its index. That index (minus one) is the missing element. This algorithm works in logarithmic time: \(O(\log n)\).
1.2 Search with a “Clue”

1.2.1 Problem

Let \( n \geq 1 \) be an integer and let \( x \) be an unknown integer that is greater or equal to 1 and less or equal to \( n \) (\( 1 \leq x \leq n \)). The goal is to find \( x \) with as minimum as possible comparisons of the type: \( x \leq i \) for \( 1 \leq i \leq n \).

In each of the following three cases, give a modification of binary search with a performance better than \( \log_2 n \) in the worst case, then compute the exact worst case number of comparisons asked by your modification as a function of \( n \).

1. \( x \) is an odd number.
2. \( x = y^2 \) is a square number.
3. \( x = 2^y \) is a power of 2.
1.2.2 Hints

1. Begin by assuming that $n = 2^k$ is a power of 2.
2. Begin by assuming that $n = (2^k)^2$ is a square of a power of 2.
3. Begin by assuming that $n = 2^{2^k}$ is a power of 2 and the exponent is also a power of 2.
1.2.3 Solution

1. Do a binary search in the space of odd numbers $< n$. (Find the number $j \leq (n/2)$ such that $2j - 1 = x$ by doing a binary search in the space $1 \leq j \leq (n/2)$ where each comparison compares $x$ with $2j - 1$ for the candidate $j$). Requires $\lceil \log_2(n/2) \rceil = \lceil \log_2 n \rceil - 1$ comparisons.

2. Identify $x$ by finding $y$ with a binary search in the space $1 \leq y \leq \sqrt{n}$ where each comparison compares $x$ with $y^2$ for the candidate $y$. Requires $\lceil \log_2 \sqrt{n} \rceil = \lceil (\log_2 n)/2 \rceil$ comparisons.

3. Identify $x$ by finding $y$ with a binary search in the space $1 \leq y \leq \log_2 n$ where each comparison compares $x$ with $2^y$ for the candidate $y$. Requires $\lceil \log_2 \log_2 n \rceil$ comparisons.
1.3 Pairs of nuts and bolts

1.3.1 Problem

Consider the following “comparison” model. There are \( n \) nuts and \( n \) bolts for \( n \geq 1 \). All nuts are of different size and each nut matches exactly one bolt. The sizes of the nuts and the bolts are so close that it is impossible to distinguish between any two nuts or any two bolts. However when matching a nut with a bolt it is clear if they match, if the nut is bigger, or if the nut is smaller. But not how much bigger or how much smaller.

Assume that the nuts and the bolts are labelled and that you “remember” all previous results and may conclude any consistent partial order that “agrees” with these results.

1. Describe an \( O(n^2) \) simple algorithm that sorts all the \( n \) pairs.
   State and prove the exact worst-case complexity of this algorithm?

2. Describe an efficient algorithm that finds the largest pair.
   State and prove the worst-case number of matchings performed by this algorithm.

3. Show that no algorithm can guarantee finding the largest pair with less than \( 2n - 2 \) matchings.

4. Describe a “better” than \( \Theta(n^2) \) algorithm that sorts all the \( n \) pairs.
   State and prove the complexity of the algorithm.
1.3.2 Hints

1. Try to adapt insertion sort.

2. An algorithm is possible which requires no more than $2n - 2$ matchings.

3. Try defining an adversary strategy which establishes the order depending on the algorithm matches. For example, the adversary may select the new nut and/or bolt whenever a matching is between a nut and/or bolt that did not participate in a matching before.

4. Try adapting Quicksort.
1.3.3 Solution

1. Do an adapted insertion sort. “Seed” the ordering with one pair by picking a random nut and trying to match it with all of the bolts until its match is found. Then, for each other nut, place it among the ordered pairs by comparing it to successive already-ordered bolts, then try to match it with all of the un-ordered bolts until its match is found.

In the worst case, to add the $i$th pair to the ordering requires $i - 1$ matchings to place the nut in the ordering, and $n - i + 1$ matchings to find its corresponding bolt, for a total of $n$ matchings for each pair, exactly $n^2$ matchings in all.

2. Begin with all nuts and all bolts classified as “possible largest”. Begin by trying to match a random nut with a random bolt. If the nut is larger, discard the bolt and replace it with another bolt from the possible bolts. If the bolt is larger, discard the nut in the same fashion. (Note that there will always be at least one remaining possible largest nut or bolt after discarding another one since the largest cannot be discarded.) If at any point the current pair are the last remaining possible nut and bolt under consideration, then they are known to be the largest pair (before trying to match them); otherwise continue matching.

If a pair is found to be a match, then, if there is only one other nut or bolt under consideration, then they are known to be the largest; otherwise one of them should be set aside and noted while the other one continues to be matched with other items from the complementary set. (This continues until an item is found to be larger than one in the pair, at which point they are both discarded, or there is only one item left besides that pair, at which point that pair is known to be the largest.)

This algorithm requires exactly $2n - 2$ matchings: the total number of items which are not part of the largest pair. In general (when there is not a match) one such item is eliminated from consideration by each matching. When there is an exact match, however, an item is not eliminated, but this is “made up for” by either: (A) both of the matched items being eliminated on a single matching when another nut or bolt is found to be larger than the one in the matched pair; or (B) the algorithm ending one matching “early” because there is only one other nut or bolt which has not been explicitly eliminated from consideration, so the matched pair is known to be the largest.

3. Say an adversary maintains the following data structure:

- $\mathcal{A}$: the set of all nuts which could be largest
- $\mathcal{B}$: the set of all nuts which could not be largest
- $\mathcal{C}$: the set of all bolts which could be largest
- $\mathcal{D}$: the set of all bolts which could not be largest

Initially $|\mathcal{A}| = |\mathcal{C}| = n$ and $|\mathcal{B}| = |\mathcal{D}| = 0$

At the end $|\mathcal{A}| = |\mathcal{C}| = 1$ and $|\mathcal{B}| = |\mathcal{D}| = n - 1$

It adopts the following strategy ($(A, C)$ means try to match nut $A$ from set $\mathcal{A}$ with bolt $C$ from set $\mathcal{C}$):

- $(A, C), |\mathcal{A}| > 1 \Rightarrow C$ too large for $A$, $A$ moved from $\mathcal{A}$ to $\mathcal{B}$
- $(A, C), |\mathcal{A}| > 0 \Rightarrow A$ too large for $C$, $C$ moved from $\mathcal{C}$ to $\mathcal{D}$
- $(A, D) \Rightarrow A$ too large for $D$ if consistent; otherwise $D$ too large for $A$, $A$ moved from $\mathcal{A}$ to $\mathcal{B}$
- $(B, C) \Rightarrow C$ too large for $B$ if consistent; otherwise $B$ too large for $C$, $C$ moved from $\mathcal{C}$ to $\mathcal{D}$
- $(B, D) \Rightarrow$ Any consistent answer

Now, it is clear that by pursuing this strategy, the adversary guarantees that with one comparison the size of either set $\mathcal{A}$ or set $\mathcal{C}$ (but not both) may be reduced by at most 1. Since no correct algorithm can be completed until the size of both of those sets has been reduced from $n$ to 1, this strategy guarantees that at least $2n - 2$ matchings are required to find the largest pair.
4. Choose a random nut as the “pivot” nut. Try all bolts with this nut, finding the matching one and dividing the rest into a larger group and a smaller group. Next try all of the remaining nuts with the bolt that matches the pivot nut (the pivot bolt), breaking them into a larger group and smaller group as well. Finally, apply this same procedure recursively on the smaller nuts and smaller bolts, and on the larger nuts and larger bolts (the base case being when the groups of nuts and bolts are each of size 1 or 0, since those are of course already sorted).

In the worst case (when the pivot pair is always the largest or smallest of the available items) the matching complexity of this algorithm is $O(n^2)$ (specifically $\sum_{r=2}^{n} (2r - 1) = n^2 + 1$). In the best (and average, among all permutations) case, the complexity is $O(n \log n)$, as can be seen from the analysis of Quicksort.
1.4 Matrix with Some Ordering

1.4.1 Problem

Let $M$ be an $n \times n$ matrix containing $n^2$ distinct integers in which the integers of each row are sorted in an increasing order (from left to right) and the integers of each column are sorted in an increasing order (from top to bottom).

1. Let $x$ be one of the numbers in $M$. Describe an efficient algorithm to find the position of $x$ in $M$.
2. What is the worst case number of comparisons made by this algorithm as a function of $n$? Justify.
3. Show an $\Omega(n)$ lower bound for the worst case number of comparisons made by any algorithm. Justify.
1.4.2 Hints

1. • Hint 1: An algorithm is possible with time complexity $O(n)$.
   • Hint 2: Begin by comparing $x$ to an element in the matrix for which the result will definitively indicate the direction (from that element) where $x$ can be found.

2. (No hint)

3. Try to construct a case where $x$ could be in any of $n$ positions completely independently of each other.
1.4.3 Solutions

1. Begin by comparing \( x \) to the bottom-left element of \( M \). If \( x \) is less than this element, move up one element and repeat. If \( x \) is greater than this element, move to the right. Continue until \( x \) is found.

2. In the worst case, when \( x \) is the top-right element of \( M \), this requires moving all the way up and all the way to the right (in some order). If each comparison is considered to reveal whether one of the numbers is greater than, less than, or equal to the other number, this is in total \( 2n - 2 \) comparisons. If it requires two comparisons to distinguish between these three possibilities, the worst-case number of comparisons is \( 4n - 3 \). (Only one comparison is needed to distinguish between the last two possibilities since \( x \) is known to be an element of \( M \).) In any case, the complexity is \( O(n) \).

3. Consider the case where \( x \) is one of the elements in the diagonal extending from the bottom-left to the top-right of \( M \), all of the elements above (or to the left of) this diagonal are less than \( x \), and all of the elements below (to the right of) it are greater than \( x + n \). In this case, \( x \) could be in any of the \( n \) positions on the diagonal, and no comparison can eliminate more than one of these possibilities, meaning that any algorithm could have to check up to all \( n \) of them, so there is a lower bound of \( \Omega(n) \) comparisons in the worst case.
1.5 Two Largest / Two Smallest

1.5.1 Problem

Let $A$ be an arbitrary (un-sorted) array of $n$ ($4 \leq n$) distinct (very large) positive integers. Describe an efficient algorithm that finds the 2 largest numbers and the 2 smallest numbers in $A$.

What is the exact worst-case number of comparisons made by this algorithm? Justify.
1.5.2 Hint

Combine the ideas of the “tournament” (parallel) method of finding the largest and second-largest element with the method of finding the largest and smallest using a search among “winners” and a search among “losers” after one round of comparisons.
1.5.3 Solution

First, perform the first round of a “tournament” by performing each element of $A$ with one other element, in effect separating them into $n/2$ candidates for largest element and $n/2$ candidates for smallest (with $n/2$ comparisons).

Find the minimum using a parallel method on the smaller half, and the maximum similarly on the larger half ($((n/2) - 1)$ comparisons for each, $n - 2$ total).

Find the second largest using by any method among “losers” to the maximum including in the first (preliminary) round. Use the same method to find the second smallest. ($[\log_2 n] - 1$ comparisons each, $2[\log_2 n] - 2$ total.)

All together, this adds up to $(3n/2) + 2[\log_2 n] - 4$ comparisons.
1.6 Majority in an Array

1.6.1 Problem

Let $A$ be an array containing $n$ very large positive integers not necessarily distinct. The array is not sorted and you cannot use “radix-sort” like algorithms to sort it. A majority is a number that appears at least $\lceil (n + 1)/2 \rceil$ times in the array (note that there can be at most one majority).

Describe an $O(n)$-time algorithm that finds a majority in $A$ if there is one. Explain why the algorithm is correct and why it has time complexity $O(n)$. 
1.6.2 Hint

Consider other using algorithms for other problems over unsorted arrays which are known to have linear complexity.
1.6.3 Solution

Find the *median* element of $A$ (which clearly must be the majority if there is one), then simply count the number of times this element occurs in $A$. If it is at least $\lceil (n + 1)/2 \rceil$, then that element is the majority, otherwise there is no majority.

Since both of these steps have time complexity $O(n)$, so does the algorithm as a whole.
1.7 Element Same as Index

1.7.1 Problem

1.7.2 Hint

Consider the effect of the condition that the elements of the sorted array be both integers and distinct.
1.7.3 Solution

Use a binary search approach. (Because the elements are distinct integers and appear in increasing order, if $A[i] > i$, for example, an element equal to its index could only occur at a position less than $i$.) The complexity of this algorithm is $O(\log n)$ since each step reduces the size of the problem by half.
1.8 Cycles in an Array

1.8.1 Problem

Let \( A = [A[1], A[2], \ldots, A[n]] \) be an array containing all the numbers \( 1, \ldots, n \) (\( A \) represents a permutation). A sequence of indices \( \langle i_1, i_2, \ldots, i_k \rangle \) is a cycle of size \( k \) if

\[
\]

Example: If \( A[i] = i \) then the singelton sequence \( \langle i \rangle \) is a cycle of size 1.

Example: In the array \([4, 6, 3, 5, 8, 7, 2, 1] \) the sequence \( \langle 1, 4, 5, 8 \rangle \) is a cycle of size 4. The other 2 cycles are \( \langle 2, 6, 7 \rangle \) and \( \langle 3 \rangle \).

Design an efficient algorithm that finds the number of cycles in \( A \). What is the running time of your algorithm? Justify your answer.
1.8.2 Hint

In how many cycles can a given element appear?
1.8.3 Solution

Start with the first element and go the element indexed by its value, continuing to do this with each visited element in turn. Increase the count of cycles (which of course begins at zero) if this ever returns you to the element where you started. This is simply detecting a cycle by following it.

“Mark” elements (indices) in some manner whenever you visit them (such as with a boolean array of $n$ elements) since you need not visit consider them again, as any element may be a part of at most one cycle. Also keep track of the index of the smallest unvisited element, advancing it to the next unmarked index whenever that element is visited. When a cycle is detected, begin “traversing” the array from the smallest unvisited element. Finish when all elements have been visited (the smallest unvisited element goes out of range) and return the cycle count.

This algorithm runs in linear time ($O(n)$) since it “leaves” each element no more than once (going to the element indexed by its value), and it advances the smallest unvisited element variable $n$ times as well.
1.9 Find Non-Max Non-Min Numbers

1.9.1 Problem

Let $A$ be an array of $n \geq 5$ distinct positive integers that may be very large compared to $n$. For each one of the following three tasks describe an efficient algorithm that tries to minimize the number of comparisons required to accomplish the task in the worst case. (Words and/or illustrations are better than code.) What is the number of comparisons each algorithm uses in the worst case? Explain.

1. Find a number in the array that is neither the maximum nor the minimum.

2. Find a number in the array that is neither the maximum and the second maximum nor the minimum and the second minimum.

3. For $1 \leq k < n/2$, find a number in the array that is neither among the top $k$ numbers nor among the bottom $k$ numbers.

Distinct numbers: The question is well defined since the numbers are distinct and $n \geq 5$. 
1.9.2 Hints

You do not need to “touch” all the $n$ numbers in order to solve the three tasks.
1.9.3 Solution

1. Compare the first and second elements, then the first and third elements, then the second and third elements. Return whichever of the three was larger in one comparison and smaller in the other.

   This always requires exactly 3 comparisons, so it has time complexity $O(1)$.

2. Compare all of the first five elements to each other (a total of 10 comparisons). Return the middle element (the one that was larger in exactly 2 comparisons and smaller in exactly 2 comparisons).

   This always requires exactly 10 comparisons, so it has time complexity $O(1)$.

3. Use the deterministic k-Selection technique to find the $(k + 1)$-smallest element. That algorithm is known to run in $O(n)$ time.
1.10 Sum from Sorted Arrays

1.10.1 Problem

**Input:** A positive integer $x > 0$ and two sorted arrays $A$ and $B$ each contains $n$ positive integers:

\[
0 \leq A[1] \leq A[2] \leq \cdots \leq A[n] \\
0 \leq B[1] \leq B[2] \leq \cdots \leq B[n]
\]

**Objective:** Find two numbers $A[i]$ and $B[j]$ such that $A[i] + B[j] = x$.

Note that $x$ and the numbers in both arrays could be very large.

Describe an algorithm which outputs “Yes” if there exist indices $1 \leq i, j \leq n$ for which $A[i] + B[j] = x$ and “No” otherwise. What is its worst-case complexity (number of operations)? Explain.
1.10.2 Hint

Consider structuring the algorithm in such a way that each comparison will eliminate one element (from $A$ or $B$) from consideration for being one of the addends.
1.10.3 Solution

Begin by indexing into $A$ at its first element and into $B$ at its last element. At each step, add the two indexed elements and compare them to $x$. If they are equal, the answer is “Yes”. If $x$ is greater, decrement the index into $B$. If the sum of the elements is greater than $x$, increment the index into $A$. If either of the elements goes out of range, the answer is “No”.

This algorithm is $O(n)$. (There can be at most $2n - 1$ comparisons before one of the indices must have gone out of range.)
Chapter 2

Graph Problems
2.1 Is Graph Bipartite?

2.1.1 Problem

Let $G$ be a simple graph with $n$ vertices and $m$ edges. Describe an efficient algorithm that determines if $G$ is bipartite or not. What is the running time of this algorithm? Justify.
2.1.2 Hint

Think about properties of graphs which are equivalent to being bipartite.
2.1.3 Solution

Follow the BFS procedure for traversing $G$, however when seeing already-visited neighbors ("children") of a vertex $v$, make sure that the level (depth in the BFS tree) of those vertices is of opposite parity to that of $v$. If the procedure finishes and this is always true, the graph is bipartite. If there are ever found to be neighbors with levels of the same parity, the graph is not bipartite.

This is true for the following equivalent reasons:

1. The property of being bipartite is equivalent to being 2-colorable. The BFS procedure can be thought of as "coloring" vertices with even level one color, and vertices with odd level another color, so having two vertices with the same parity level next to each other means the graph is not 2-colorable.

2. Equally, a bipartite graph is one with no odd cycles, and reaching an already-visited node in the BFS procedure with the same parity as the current vertex, its neighbor, can be thought of as closing an odd cycle (formed from the edges of the BFS tree and the edge between those two vertices).

This algorithm has the same time complexity as BFS traversal: $O(n + m)$ if the graph is represented with an adjacency list and $O(n^2)$ if an adjacency matrix is used.
2.2 Maximum/Maximal Independent Set in a Graph

2.2.1 Problem

An independent set in a graph is a set of vertices such that there is no edge between any pair of vertices in the set.

A maximum independent set in a graph is an independent set that has the largest number of vertices among all possible independent sets.

A maximal independent set in a graph is an independent set that is not contained in another independent set. That is, any new vertex that would be added to the set would create a non-independent set.

1. Find the maximum independent set in a complete bipartite graph that has $x$ vertices in one side and $y$ vertices in the other side for $x + y = n$. What is the size of this set?

2. Find the smallest possible maximal independent set in a complete bipartite graph that has $x$ vertices in one side and $y$ vertices in the other side for $x + y = n$. What is the size of this set?

3. Finding a maximum independent set is an NP-Complete problem whereas finding a maximal independent set is not. Describe an efficient (greedy) algorithm that finds one of the maximal independent sets in a graph. What is the worst case running time of your algorithm if the graph is maintained with adjacency lists? What is the worst case running time of your algorithm if the graph is maintained with adjacency matrix? Explain.
2.2.2 Hints
2.2.3 Solution

1. \( \max(x, y) \)

2. \( \min(x, y) \)

3. Begin with all vertices as candidates for the maximal independent set. Add a random vertex and throw out all of its neighbors as candidates. Continue until there are no more candidates. This algorithm has time complexity \( O(n + m) \) if the graph is maintained as an adjacency list and \( O(n^2) \) if an adjacency matrix is used, with finding the neighbors being the dominating step in both cases.
2.3 Pseudocode on Graph

2.3.1 Problem

The following algorithm is applied to a simple undirected graph with \( n \) vertices in the set \( V \).

\[
\text{let } S \text{ be a set containing an arbitrary vertex } v \in V: S = \{v\} \\
\text{let } R \text{ be the set off all the neighbors of } v: R = \{\text{neighbors}(v)\} \\
\text{while } R \text{ is not empty do} \\
\quad \text{let } v \in R \text{ be an arbitrary vertex from } R \\
\quad \text{add } v \text{ to } S: S = S \cup \{v\} \\
\quad \text{omit } v \text{ from } R: R = R \setminus \{v\} \\
\quad \text{omit from } R \text{ all vertices that are not neighbors of } v: R = R \setminus \{\text{nonneighbors}(v)\}
\]

What could you say about set \( S \) when the algorithm terminates? Justify.
2.3.2 Hint

What effect do the steps inside the loop have on the relationship among elements of $S$?
2.3.3 Solution

$S$ is a clique. Initially, the sole vertex in $S$ is adjacent to all vertices in $R$, and this is maintained by excluding all vertices from $R$ which are not adjacent to each vertex added to $S$. 
2.4 Removing One-Edge Vertices

2.4.1 Problem

Let $G$ be a simple undirected graph with $n$ vertices and $m$ edges. Apply the following process on $G$:

- As long as $G$ contains at least one vertex with degree 1 (exactly one neighbor), remove one of these vertices and the edge that contains it from $G$.

For which graphs does this process terminate with a graph that has exactly one vertex? Explain.
2.4.2 Hint
2.4.3 Solution

The process terminates with a one-vertex graph if the input graph is a non-null tree. If the graph has a cycle, then the cycle will “sustain itself” (each vertex will always have degree of 2 attributable to the cycle and thus cannot be removed), and if the graph is a (non-tree) forest, it will end with one isolated vertex for each connected component.
2.5 Set-Generating Algorithm from Graph

2.5.1 Problem

The following algorithm is applied to a simple undirected graph.

\[
\begin{align*}
&\text{let } S \text{ be the empty set;} \\
&\text{let } R \text{ be the set of all vertices;} \\
&\text{while } R \text{ is not empty do} \\
&\quad \text{let } v \text{ be an arbitrary vertex from } R; \\
&\quad \text{add } v \text{ to } S; \\
&\quad \text{remove } v \text{ from } R; \\
&\quad \text{remove from } R \text{ all the neighbors of } v \text{ that are in } R; \\
&\text{end-while}
\end{align*}
\]

What can you say about the set \( S \) when the algorithm terminates? Explain.
2.5.2 Hint
2.5.3 Solution

$S$ is both a maximal dominating set because each time a vertex was added to $S$ all of (and only) its neighbors were removed from consideration, and a dominating set because all of the graph’s vertices are either in $S$ or a neighbor of a vertex in $S$. 
2.6 Analyzing Classes of Graphs: Forests and Trees

2.6.1 Problem

Let $G$ be a simple undirected graph with $n \geq 1$ vertices and $m$ edges. A forest is a graph with no cycles. A tree is a connected forest. Consider the following nine classes of graphs:

1. Connected graphs with $m = n - 1$ edges.
2. Bipartite graphs with $m = n - 1$ edges.
3. Graphs for which between any pair of vertices there exists at most one path.
4. Graphs with no cycles for which adding any edge would create exactly one cycle.
6. Graphs with no triangles.
7. Graphs that can be colored with at most two colors.
8. Connected graphs for which omitting any edge would disconnect them.
9. Connected graphs for which the BFS traversal tree is identical to the DFS traversal tree when both traversals start with the same vertex and both traversals consider the same order on the vertices.

Which one of the following three statements is true for each one of the above nine classes of graphs? Justify.

1. All the graphs in this class are trees.
2. All the graphs in this class are forests but not all of them are trees.
3. Some graphs in this class are not forests.
2.6.2 Hints
2.6.3 Solution

1. All trees. A cycle requires the same number of edges as vertices, which would not leave enough edges to connect all of the vertices, a contradiction. Since any graph in this class is both connected (given) and acyclic, it is a tree.

2. Some are not forests, for example a graph which consists of a 4-cycle and an isolated vertex.

3. All forests, but not all trees. These graphs are all acyclic, because there are necessarily multiple paths between any vertices in a cycle (proceeding the cycle in opposite directions). They are not necessarily all connected however: the graph consisting solely of 2 isolated vertices, for example, is a forest but not a tree.

4. All trees. These graphs are acyclic as per the definition of the class, and they are also connected because otherwise one could connect two of the connected components by adding an edge which would not create a cycle, a violation of the class definition.

5. All forests, but not all trees. These graphs are all acyclic, because a cycle cannot exist without edges, but they are not all connected: any graph in this class with 2 or more vertices is an unconnected forest consisting solely of multiple isolated vertices.

6. Not all forests. Any triangle-free graph containing a cycle larger than 3 (such as the graph consisting solely of the 4-cycle) is neither a tree nor a forest.

7. Not all forests. The 4-cycle is an example of a bipartite graph which is neither a tree nor a forest.

8. All trees. These graphs are connected by definition, and they can be seen to be acyclic because otherwise one could omit one of the edges in the cycle and this would not disconnect the graph (because any vertices which were connected by that edge remain connected by the edges which constitute the remainder of the former cycle).

9. All forests but not all trees. These graphs can be seen to be acyclic because the DFS and BFS procedures create different structures upon reaching a cycle. In particular, BFS reaches vertices on either side of the “entry point” in the cycle before the rest of the cycle while DFS does the opposite. They need not be connected however, for example BFS and DFS create the same structure for the graph consisting solely of 2 isolated vertices.
2.7 Identify a Celebrity

2.7.1 Problem

**Story:** A celebrity among a group of \( n \) people is a person who knows nobody but is known to everybody else. The task is to identify a celebrity by only asking questions to people of the form: “Do you know him/her?”

1. Model the story with a directed graph that is represented with an adjacency matrix. In particular, answer the following questions:

   (a) Who are the vertices of this graph?
   
   (b) What is the meaning of a directed edge in this graph?
   
   (c) What is a celebrity?
   
   (d) What is the equivalent in graph terms to the defined question?
   
   (e) What is the complexity objective?
   
   (f) Can a group have more than one celebrity?

2. Design an efficient algorithm to identify a celebrity or determine that the group has no such person.

   How many questions does this algorithm need in the worst case? Justify.
2.7.2 Hint

Think about exactly what information a single edge provides toward answering this question. (Some of this information may be duplicated by multiple edges.)
2.7.3 Solution

1. (a) The vertices represent the people in the group.
   (b) A directed edge indicates that the person represented by the source vertex “knows” the person represented by the destination vertex.
   (c) A celebrity is an edge with in-degree \( n - 1 \) and out-degree zero.
   (d) Does edge \((v_1 \rightarrow v_2)\) exist?
   (e) The objective is to reduce the number of “questions asked” (matrix accesses) as a function of \( n \).
   (f) No. Since a celebrity must be “known” by everyone else in the group, and know no one, if one celebrity exists, another person could not be a celebrity because that person could not be known by the existing celebrity.

2. Begin with all \( n \) people as candidates to be a celebrity. Ask a random person, \( x \) if he knows random person \( y \). If so, eliminate \( x \) as a candidate to be a celebrity. If not, eliminate \( y \) as a candidate to be a celebrity. Continue in this manner until, after \( n - 1 \) questions, there is only one candidate to be a celebrity, then check with each of the other \( n - 1 \) people to confirm that they all know that person (at most \( n - 1 \) questions), and with the person to confirm that she does not know any of the other people (also at most \( n - 1 \) questions). Thus, it takes at most \( 3n - 3 \) questions to identify a celebrity if such a person exists (order \( O(n) \)).
2.8 Clique / Triangle

2.8.1 Problem

Let $G$ be a simple undirected graph with $n \geq 1$ vertices and $m$ edges.

A clique in $G$ is a set of vertices such that all possible edges between any pair of vertices in the clique exist. A maximum clique in $G$ is a clique that has the largest number of vertices among all the cliques of $G$.

1. If $G$ does not have a clique of size $k$, can $G$ have a clique of size $k + 1$? Explain.

2. What is the size of the maximum clique in a bipartite graph? Explain.

3. Describe an $O(n^k)$-time algorithm to find a clique of size $k$ in $G$ if such a clique exists. Use simple words and do not write program codes. Explain why the complexity of your algorithm is $O(n^k)$.

4. Describe an algorithm to find a triangle (a clique of size 3) in $G$ if exists. Your algorithm should have a better than $O(n^3)$-time complexity (for example, $O(nm)$ is better than $O(n^3)$). Simple words are preferable to writing program code. Justify your complexity claim.
2.8.2 Hints
2.8.3 Solution

1. No. Every clique of size $k + 1$ contains $k + 1$ cliques of size $k$ within it (which can be created by excluding a vertex and all edges that include it).

2. No vertices in a clique could be in the same partition of a bipartite graph since all vertices in a clique are connected to each other, thus there can be no clique of size greater than 2 by the pigeonhole principle.

3. Simply consider all combinations of $k$ vertices and check whether or not all possible edges between them exists. There are $n^k$ such combinations (including those with repetitions of the same vertex, which by definition do not form a clique of size $k$).

4. For each edge check whether it is part of a triangle with each of the other $n - 2$ vertices. This is order $O(nm)$ for adjacency lists if the existence of the two edges for each of these potential triangles can be checked in constant time (feasible depending on the metadata maintained at the node of the linked list of vertices).