Michael Inoue
405171527

Ling
Discussion 1C 2-3:50PM

CS 180

Homework 2

16 October 2019

1 Solution

Extend the topological ordering algorithm so that, given an input directed graph G, it outputs one of two things: (a) a topological ordering, thus establishing that G is a DAG; or (b) a cycle in G, thus establishing that G is not a DAG. The running time of your algorithm should be $O(m+n)$ for a directed graph with n nodes and m edges.

The algorithm below attempts to solve this problem.

Algorithm 1: Determines if a graph G is a DAG

```
Input  : Graph G with nodes $v \in V$ and edges $e \in E$
Output: Topological Ordering or Cycle
Initially queue of nodes with no active incoming nodes $S$ is empty
Initially the number of edges $e_n$ associated with each node $v$ is 0
Initially the list of nodes $L$ pointed to by each $v$ is empty
Initially the total number of nodes $n$ is 0
Initially the topological ordering list $T$ is empty
Initially the active nodes list $A$ is empty
foreach edge $e$ do
    Choose such an edge $e = (v, w)$
    Add $e$ to $v$'s queue $I$
    Increment $w$'s edge count $e_n$ by 1
end
foreach node $v$ do
    Choose such a node $v$
    Increment total number of nodes $n$ by 1
    Add $v$ to $A$
    if $v$'s edge count $e_n$ is 0 then
        Add $v$ to $S$
    end
end
while $S$ is not empty do
    Dequeue node in front of $S$, $w$
    Remove $w$ from $A$
    Add $w$ to topological ordering $T$
    Consider each edge $e = (w, u)$ in $w$'s list $L$
    Decrement $u$'s number of edges $e_n$ by 1
    if $u$'s number of edges $e_n$ is 0 then
        Add $u$ to $S$
    end
    Decrement total number of nodes $n$
end
if $n$ is 0 then
    return topological ordering $T$
else
    Initially list of nodes that form cycle $C$ is empty
    Initially mark each active node as unvisited
    Choose an arbitrary active node $v \in A$
    Mark $v$ as visited
    Let boolean flag nocycle be true
    while nocycle do
        Consider the first edge $e = (v, u)$ in $v$'s list
        if $u$ is visited then
            Set nocycle to false
        else
            Mark $u$ as visited
            Add $u$ to cycle $C$
            Set $v = u$
        end
    end
    return cycle $C$
end
```

We first prove the correctness of our algorithm. In our algorithm, in lines (1-12), we simply observe which nodes are valid candidates to be considered sources, that is, nodes with no current precedence relationships, or more specifically, nodes with no incoming edges. In lines (12-19) we add such nodes to a queue of source nodes, S. Now, in lines (20-29), we simply output each source node and remove the edges of the nodes dependent on it. If there were any nodes that were dependent on v that no longer have any edges, we consider them from that point valid source nodes and add them to S. This is a valid procedure, as source nodes don't require any other nodes to be outputted, and once they are outputted, the nodes that were dependent on them are no longer dependent. We then analyze our remaining nodes and repeat this procedure, until S is empty.

Now, we learned in class and in the book that if we have active nodes remaining, that is, nodes that were not deleted from our procedure, but no source nodes left, then we must have a cycle in our graph. If not, then we must have output every node in order of dependence based on our setup, meaning our topological sort is valid.

Now, in lines (33-49), as mentioned earlier, if we have active nodes remaining and each has an incoming edge (e.g. is not a source), then we must have a cycle in our graph. We simply perform a variant of Breadth-First Search (BFS) on our graph, marking each node as visited once we stop at said node in our graph, outputting said node in our cycle C, and traversing an arbitrary edge of each node. Because our remaining graph contains a cycle, and BFS will reach every node (as all have incoming edges), at some point we must stop at a node that we already visited before, in which case, our output C must be, by definition, a cycle. Thus, we conclude that our algorithm yields correct results, either a proper topological ordering or a cycle.

Time Complexity:
This algorithm seeks to determine if a graph G is a directed acyclic graph (DAG) or a directed cyclic graph (DCG). If a graph is directed, then the algorithm returns a topological ordering T of the nodes in G. If not, then it returns a cycle.

We note that an edge $e = (v, w)$ in this scenario is the relationship between v and w such that v points to w.

In lines (7-11), in the for each loop, we first traverse each edge and perform a constant operation, adding e to v's queue I and incrementing w's edge count. If there are m edges this will be done in $O(m)$ time. Next, in lines (12-19), in the for each loop, we traverse each node to see if it belongs in the list of nodes with no incoming edges. If so, we add it to a queue S, which is a constant operation ($O(1)$). With n nodes, this process will take $O(n)$ steps.

Next, the algorithm removes one node w from S and remove the edges e associated with w and the nodes they point to, a list of nodes L. For each node $u \in L$, we update u such that its number of incoming edges decreases. We add u to S if e was the last edge pointing to u. We repeat this process until we run out of edges. Thus, we only traverse each edge and node once more, and this process runs in $O(m + n)$ time.

Finally, we either return the topological ordering if we have exhausted each edge and there are no nodes with incoming edges, or the remaining graph, a cycle, otherwise. If we return a topological cycle, we just need to return the cycle we already built, which is an $O(1)$ operation. If we return a cycle, we utilize a variant of Breadth-First Search to simply iterate through an arbitrary node, choosing the first edge in its list until we find the node we started with. In the worst case, we will traverse each node, so this takes $O(n)$ time.

In total, our worst case complexity $T(m, n)$ is:

$$\begin{aligned} T(m,n) &\leq O(n) + O(m+n) + O(n) && (1) \\ &\leq O(m+n) && (2) \end{aligned}$$

2 Solution

Algorithm 2: Determines if judgments are consistent

```
   Input  : Specimens of butterflies and judgments relating those butterflies
   Output: Determination of whether judgments are consistent
1  foreach butterfly $i$ do
2      Create an unlabeled node representing $i$
3  end
4  foreach judgment $x = (i, j)$ do
5      if $i$ and $j$ are labeled the same then
6          Create an edge $s$ from $i$ to $j$ indicating they are the same
7      end
8      if $i$ and $j$ are labeled as different then
9          Create an edge $d$ from $i$ to $j$ indicating they are different
10     end
11 end
12 Choose some arbitrary butterfly node $b$ for each component of our current graph
13 Give $b$ some label $l$
14 Initially a set of levels (with a level being a set of nodes) $L = \{L_0, L_1, ...\}$ is uninitialized
15 Add $b$ to $L_0$
16 foreach node $b'$ at $L_i$, for $i = 0, 1, 2, ...$ do
17     Consider each edge $(b', b'')$ from $b'$ to $b''$
18     if $b''$ is unlabeled then
19         Add $b''$ to $L_{(i+1)}$
20         if $b''$ is the same as $b'$ then
21             Label $b''$ as the same species as $b'$
22         end
23         if $b''$ is different than $b'$ then
24             Label $b''$ as the opposite species of $b'$
25         end
26     end
27 end
28 foreach judgment $x = (i, j)$ do
29     if $i$ and $j$ are labeled the same and $i$ and $j$ have different labels then
30         return results are inconsistent
31     end
32     if $i$ and $j$ are labeled as different and $i$ and $j$ have the same label then
33         return results are inconsistent
34     end
35 end
36 return results are consistent
```

We first prove the correctness of our algorithm. In lines (1-11), we treat each butterfly as a node and iterate through each judgement to create edges between butterflies mentioned in said judgment, with the edges indicating whether they are the same or different.

Next, in lines (12-27), we perform a Breadth-First Search (BFS) starting at some arbitrary butterfly node, b. Without loss of generality, we can assume that b is a particular species. Based on the properties of the BFS tree we constructed and our judgment list, from b we can label every other node b' if there exists some path between b and b', that is, we should be able to construct a graph that will reach b' from b (thus, implicitly relating the species of b and b') and label every species reachable from b. If no such edges exist between the nodes (e.g. there is no set of judgments relating the two or said judgments were arbitrary), then we need not worry about comparing b to b', as there is no relevant set of judgments to relate them regardless. Thus, we can claim that if our BFS labeling is consistent, then our judgments are consistent, and that if our BFS labeling is inconsistent, then our judgments are inconsistent.

Indeed, our algorithm checks for the consistency of our labeling in lines (28-35) when iterating through each judgment and checking the labeling of each corresponding nodes: if our judgment suggests that two nodes x and y should be the same and their labels are different, then we return that our results are inconsistent. On the other hand, if no such nodes x and y exist, then as previously mentioned, we know that the judgments were consistent. Our algorithm accounts for this by returning that the results are consistent at line (36) after we check every judgment.

Thus, we conclude that our algorithm yields the correct judgment on the consistency of our judgments.

Complexity Analysis:

Lines (1-11): Our algorithm first constructs a graph $G = (V, E)$ with nodes V and edges E, where each butterfly is a node, and each judgment between two butterflies is an edge. There are n nodes and m edges, so constructing this graph is $O(m + n)$.

Lines (16-27): Next, our algorithm utilizes a Breadth-First Search (BFS) Algorithm to label nodes at some level away from an arbitrary node b, which we define in lines (12-15) to have some arbitrary label l. In the for loop, we iterate through each node b' once, and consider each edge incident to b'. Let $n_{b'}$ denote the degree of b'. The time that it takes to iterate through each node of b' is $O(n_{b'})$. For each node $b' \in V$, the total time is

$$O(\sum_{b' \in V} n_{b'}).$$

But

$$\sum_{b' \in V} n_{b'} = 2m, \tag{3}$$

so we iterate through each edge twice in the worst case, updating the labels of unlabeled nodes relative to their incident nodes, until every node is labeled. Because we iterate through each node once in the worst case for each component and each edge twice, the time for this process is $O(m + n)$.

Lines (28-36): Finally, our algorithm iterates through each judgment once more, utilizing the labels provided by lines (16-27) of the algorithm to determine whether judgments are consistent. Because there are m judgments, and we iterate through each judgment once, this process is $O(m)$.

Adding our worst-case complexities together for each process of our algorithm, we have our total worst-case time complexity for our algorithm, $T(m, n)$:

$$\begin{aligned} T(m, n) &\leq O(m + n) + O(m + n) + O(m) \quad (4) \\ &\leq O(m + n) \quad (5) \end{aligned}$$

3 Solution

Suppose that an n-node undirected graph G = (V, E) contains two nodes s and t such that the distance between s and t is strictly greater than $\frac{n}{2}$. Show that there must exist some node v, not equal to either s or t, such that deleting v from G destroys all $s-t$ paths. (In other words, the graph obtained from G by deleting v contains no path from s to t.) Give an algorithm with running time $O(m+n)$ to find such a node v.

The algorithm below aims to solve this problem:

Algorithm 3: Determination of which node is included in all paths between nodes s and t

```
Input  : Non-Directed Graph G with nodes $s$ and $t$ a distance $> \frac{n}{2}$ apart
Output: A node $v$ such that $v$'s deletion destroys all paths from $s$ to $t$
Mark $s$ as discovered
For all other $v \in V$ mark as undiscovered
Initialize layer L[0] to contain the single element $s$
Set the layer counter i = 0
Set the current BFS tree T = $\emptyset$
while L[i] is not empty do
    Initialize an empty list L[i + 1]
    foreach node $u \in$ L[i] do
        Consider each edge $(u, v)$ incident to $u$
        if $v$ is undiscovered then
            Mark $v$ as discovered
            Add edge $(u, v)$ to the tree $T$
            Add v to the list L[i + 1]
        end
        Increment layer counter $i$ by one
    end
end
Set layer counter $i = 1$
while $L[i]$ is not empty do
    if layer $L[i]$ contains only one node $v$ then
        return $v$
    end
    Increment $i$ by one
end
```

Proof. We now prove that our algorithm yields a node v between s and t, such that if v was deleted, all paths between s and t would be eliminated.

We consider the fact that in a graph with n nodes, the distance between s and t is strictly greater than $\frac{n}{2}$. Between s and t, we have at most $n-2$ nodes, as we exclude s and t. By utilizing a Breadth-First Search (BFS) Algorithm to construct a BFS tree rooted at s, we recall that we can find the distance between s and any node t by simply counting the layers or levels between s and t (i.e. the number of edges). Now, we note that one layer must contain only one node, as if each contained at least 2 nodes, then we would be left with at least $(\frac{n}{2})(2) + 2 = n + 2$ nodes total, which exceeds the number of nodes in our original graph. Thus, there must be a layer with only one node. Call this node v. But if this is the case, by principle of a BFS tree, the path from s to a node w in a later level than v must contain v in its path. Thus, our node t must contain v in its path starting at s, and removal of v would eliminate any such path $s-t$ from s to t.

Our algorithm does just this process, utilizing a BFS algorithm (lines 1-19) to create a representative tree with layers and simply iterating through those layers (lines 20-26) to find the layer with only one node. Thus, we conclude that our algorithm yields such a node v. $\square$

Complexity Analysis:
Lines (1-19): Our algorithm first runs a BFS Algorithm, which runs in $O(m+n)$ time for a tree with m edges and n nodes.
Lines (20-26): Next, our algorithm iterates through every layer after s to find the first layer between s and t that only contains one node. Because we iterate through at most $n-2$ nodes total, this runs in $O(n)$ time.
Adding our time complexity together, we have our total time complexity, $T(m,n)$:

$$\begin{aligned} T(m,n) &\leq O(m+n) + O(n) && (6) \\ &\leq O(m+n) && (7) \end{aligned}$$

4 Solution

Design an algorithm that answers questions of this type: given a collection of trace data, the algorithm should decide whether a virus introduced at computer C_a at time x could have infected computer C_b by time y. The algorithm should run in time $O(m + n)$.

The algorithm below attempts to solve this problem.

Algorithm 4: Virus Detection

```
Input  : List of communication at times $(C_i, C_j, t_k)$, Infected computer $C_a$ at time $x$, Possibly
         infected computer $C_b$ at time $y$
Output: Determination if $C_b$ could be infected at time $y$
Initially list $L$ of communication is empty
Initially array of nodes $A$ is empty ($A[i]$ is null for each $i$)
foreach communication $c = (C_i, C_j, t_k)$ in input list, in sequential order do
    if $t_k \geq x$ and $t_k \leq y$ then
        Add $c$ to $L$
    end
end
Initially list of nodes $N$ is empty
foreach communication $c = (C_i, C_j, t_k)$ in $L$ do
    Create node $u = (C_i, t_k)$
    Create node $v = (C_j, t_k)$
    Add $u$ and $v$ to $N$
    Create directed edge from $u$ to $v$ and $v$ to $u$
    if $A[i]$ is not null then
        Create directed edge from node A[i] to $u$
    end
    if $A[i]$ is null and $A[i]$'s $C_i$ or $C_j$ component is $C_a$ then
        Create a node $w = (C_a, y)$ and an edge from $w$ to $(C_a, t_k)$ (this adds our infected node, $C_a$
          into the list of nodes and links to the first instance of $C_a$ in our input list so it can infect
          other nodes later on)
    end
    Set A[i] = $u$
    if $A[j]$ is not null then
        Create directed edge from node A[j] to $v$
    end
    Set A[j] = $v$
end
Initially all nodes except $(C_a, y)$ are uninfected
Initialize layer L[0] to contain the single element $(C_a, y)$
Set the layer counter i = 0
Set the current BFS tree T = $\emptyset$
while L[i] is not empty do
    Initialize an empty list L[i + 1]
    foreach node $u \in L[i]$ do
        Consider each edge $(u, v)$ coming from $u$ to $v$
        if $v$ is not infected then
            Mark $v$ as infected
            Add edge $(u, v)$ to the tree $T$
            Add v to the list L[i + 1]
        end
        Increment layer counter $i$ by one
    end
end
foreach node $n = (C_i, t_k) \in N$ do
    if $C_i$ is $C_b$ and is marked infected then
        return true
    end
end
return false
```

We first prove the correctness of the algorithm. In lines (1-25), the algorithm takes a subset of the input list and converts each triplet into a pair of nodes. First, the subset chosen is simply all connections between the time of infection x and the time of inquiry, y, inclusive. Indeed, any communication before the time of infection is irrelevant, and any communication after time y is not within the scope of the question asked. Thus, we choose all communication between x and y, inclusive as candidates for determining which computers are infected. We then build a graph such that each communication in a pair (C_i, C_j, t_k) is represented by two nodes (C_i, t_k) and (C_j, t_k) that have directed edges with each other, signifying that they can reach each other at some time t_k such that $x \leq t_k \leq y$, and if there is some other node (C_i, t'_k) that already exists in our graph, then we create an edge from (C_i, t'_k) to (C_i, t_k), and do the same process for (C_j, t_k) if there is some (C_j, t'_k) that came before it in our graph.

Next, in lines (26-41), we perform a Breadth-First Search (BFS) on our newly constructed graph, starting at our node $n' = (C_a, y)$. We note that all nodes are initially uninfected except n'. We claim that if an infected node i has an edge connecting it to an uninfected node u, then u is infected by the end of the algorithm, and thus any node reachable by n' (i.e. for some node n'' there exists a path from n' to n'') is infected by the end of the algorithm.

To prove such a claim, we prove by contradiction. Suppose, towards a contradiction, that there is some path $p = (n', ..., n'')$, and n'' is not infected by the end of the algorithm. But then, because our resulting graph is a BFS Tree, if n'' is in our resulting graph, we can traverse the edges backwards from n'' to n'. We must conclude that n' is not infected, which is a contradiction. Thus, n'' must be infected. We now wish to prove that if there is no path between n' and n'', then n'' is not infected. Suppose, towards a contradiction, that n'' is infected. Then there exists some i such that i infected n'', as by line (26), n'' is uninfected initially. We repeat this process, analyzing the edges connecting to i. We note that no node except n' can be the source of the infection, by line (26) of the algorithm. Thus, there must exists some path from n' to i. But that implies there is a path from n' to n'', as i is connected to n'', and this is a contradiction. Thus, we conclude that our algorithm correctly predicts if a computer C_a will be infected by time y after a computer C_b is infected at time x.

Time Complexity: $O(m + n)$
We now prove that the complexity is $O(m + n)$.

Lines (1-7): In the for each loop, for each communication triplet, we perform a constant operation ($O(1)$), which is adding an element to a list. There are m communication triplets, so this process is $O(m)$.

Lines (8-25): In the for each loop, for each communication triplet, we perform constant operations ($O(1)$), which is the creation of two nodes and at most two edges linking nodes. There are m communication triplets, so this process is $O(m)$.

Lines (26-41): We run a directed BFS algorithm on our nodes in N, with $2m$ edges (each m communication pair forms an edge) and n nodes at most. BFS was proven to be $O(m+n)$ in class, so this process is $O(m+n)$.

Lines (42-47): In the for each loop, for each node contained in N, we perform a constant operation ($O(1)$), which is a comparison. There are $2m$ edges, so this process is $O(m)$.

Adding our worst-case complexities together for each process of our algorithm, we have our total worst-case time complexity for our algorithm, $T(m, n)$:

$$\begin{aligned} T(m,n) &\leq O(m) + O(m) + O(m+n) + O(m) && (8) \\ &\leq O(m+n) && (9) \end{aligned}$$

5 Solution

Give an efficient algorithm to do this: either it should produce proposed dates of birth and death for each of the n people so that all the facts hold true, or it should report (correctly) that no such dates can exist—that is, the facts collected by the ethnographers are not internally consistent.

The algorithm below attempts to solve this problem.

Algorithm 5: Birth and Death Date Consistencies

```
Input  : People $P_i$, Relationships $R$
Output: Proposed birth dates and deaths, or reported inconsistencies
Initially queue of nodes with no active incoming nodes $S$ is empty
Initially the number of edges $e_n$ associated with each node $v$ is 0
Initially the list of nodes $L$ pointed to by each $v$ is empty
Initially the total number of nodes $n$ is 0
Initially the topological ordering list $T$ is empty
foreach Person $P_i$ do
    Create a node $b_i$ (birth date)
    Create a node $d_i$ (death date)
    Create a directed edge from $b_i$ to $d_i$, indicating $b_i$ is before $d_i$
end
foreach relationship $r = (P_i, P_j) \in R$ do
    if $P_i$ and $P_j$ have overlapping lives then
        Create an edge from $b_i$ to $d_j$
        Create an edge from $b_j$ to $d_i$
    end
    if $P_i$ dies before $P_j$ was born then
        Create an edge from $d_i$ to $b_j$
    end
end
foreach edge $e$ do
    Choose such an edge $e = (v, w)$
    $e$ to $v$'s queue $I$
    Increment $w$'s edge count $e_n$ by 1
end
foreach node $v$ do
    Choose such a node $v$
    Increment number of nodes $n$ by one
    if $v$'s edge count $e_n$ is 0 then
        Add $v$ to $S$
    end
end
while $S$ is not empty do
    Dequeue node in front of $S$, $w$
    Add $w$ to topological ordering $T$
    Consider each edge $e = (w, u)$ in $w$'s list $L$
    Decrement $u$'s number of edges $e_n$ by 1
    if $u$'s number of edges $e_n$ is 0 then
        Add $u$ to $S$
    end
    Decrement total number of nodes $n$
end
if $n$ is 0 then
    return topological ordering $T$
else
    return inconsistency
end
```

We first prove the correctness of the algorithm. In lines (1-19), we first create for each person P_i two nodes: b_i, their birth date, and d_i, their death date. We note that logically, a person's death date must come before their birth date. We establish an directed edge from b_i to d_i to establish this precedence relationship. Next, we iterate through each relationship (P_i, P_j). If P_i dies before P_j was born, then we create an edge from d_i to b_j to indicate the precedence relationship between P_i's death and P_j's birth. If P_i has an overlapping life with P_j, then P_i must be born before P_j dies and vice versa. Thus, we create an edge from b_i to d_j and from d_i to b_j to establish this precedence relationship.

At this point, we see that we have encoded a precedence relationship between each (P_i, P_j). We see that our problem at this point is identical to problem (1): given a list of nodes with precedence relationships, how can we output each node such that nodes with precedence come first? As in problem (1), we see this occurs if and only if there is no cycle in our constructed graph. In lines (32-42), we repeatedly remove each node with no dependencies (no incoming edges), until we are left with no nodes. As proved in (1), we can simply output these nodes in the order we remove them, or if we are at some point left with a cycle, we know that there is no ordering such that we can output any node, as each are interdependent on each other. Our algorithm accomplishes this verification in lines (33-42). Thus, we conclude that our algorithm yields a consistent ordering if possible, or reports an inconsistency if necessary.

Complexity Analysis:
Lines (1-19): We first construct a graph representing our problem, with at most 2 nodes per person and 2 edges per relationship (an $O(1)$ operation for each node and each edge). We have n persons and m relationships, so this process runs in $O(m + n)$ time.

Lines(20-46): We now perform a topological sort on our graph. As proved in problem (1), this can be done in $O(m + n)$ time.

Adding our worst-case complexities together for each process of our algorithm, we have our total worst-case time complexity for our algorithm, $T(m, n)$:

$$\begin{aligned} T(m,n) &\leq O(m+n) + O(m+n) \qquad (10) \\ &\leq O(m+n) \qquad (11) \end{aligned}$$

6 Solution

An array of n elements contains all but one of the integers from 1 to n+1.
(a) Give the best algorithm you can for determining which number is missing if the array is sorted, and analyze its asymptotic worst-case running time.

Algorithm 6: Find Missing Element in Array from 1 to $n+1$

```
Input  : Array $A$ of $n$ elements such that $A \subset S = [e_1, e_1 + 1, \dots, e_1 + n]$ and there is one $e \in S$,
         $e \notin A$, Array $B$ of $n+1$ elements such that $B = [e_1, e_1 + 1, ..., e_1 + n]$ for $e_1 \in \mathbb{N}$
Output: Missing element
1  if $n$ is 1 then
2      if $A[0]$ is $B[0]$ then
3          return $B[1]$
4      else
5          return $B[0]$
6      end
7  end
8  if $A[\frac{n}{2} - 1] > B[\frac{n}{2} - 1]$ then
9      return Algorithm 6 (Array A of $\frac{n}{2}$ elements, Array B of $\frac{n}{2} + 1$ elements)
10 else
11     return Algorithm 6 (Array A+$\frac{n}{2}$ of $\frac{n}{2}$ elements, Array B+$\frac{n}{2}$ of $\frac{n}{2} + 1$ elements)
12 end
```

(Note: the notation used to define A and B utilizes a term $e_1 \in \mathbb{N}$. This is simply to signify that this algorithm works for any sorted set starting at a fixed number e_1, which is crucial for understanding the proof of the algorithm below. Initially, we note that for our initial arrays A and B, $e_1 = 1$. Thus B = [1,2,...,n+1] and A is some subset missing an element $e \in B$, as before.

We note the following preconditions of our algorithm:
1.) Array A is a sorted array of n elements, starting at A (the zeroth address) such that there is 1 element $e \in B$, $e \notin A$.
2.) Array B is an array of $n+1$ elements, defined in our initial input as B = [1, 2, ..., n+1].
3.) Arrays start at their initial address (inclusive), and terminate at their size, the number of elements passed into the algorithm parameters (exclusive).
4.) We can add a constant displacement $c \in \mathbb{N}$ such that $A + c$ is a valid array starting at address position $A + c$ and terminating at the number of elements passed into the algorithm parameters.
5.) Array indices start at 0
6.) We define our division $\frac{n}{2}$ on integers to be $\frac{n}{2} = \lfloor \frac{n}{2} \rfloor$

We now wish to prove the postcondition of our algorithm:
1.) Our algorithm returns the missing element of the array, $e \in \{1, 2, ..., n\}$

We now prove by strong induction that our algorithm is correct.

Our base case is when $n = 1$. In this case, our array A has one element, A[0], and our array B has two elements, B[0] and B[1]. We defined B to have one more element then A, namely, the element A is lacking. If A[0] = B[0], then we know that B[0] is in the array, and thus the element we are missing is B[1]. The algorithm returns B[1] in this case, which is correct. Now, if this condition doesn't hold, then A[0] $\neq$ B[0], and thus A does not contain B[0] in this case. Thus, we know that B[0] is the missing element, so we return B[0], which is correct. Thus, our base case holds.

Now, we claim our inductive hypothesis to be that Algorithm 6 returns the correct result for A sizes 1 to $n-1 \in \mathbf{N}$ and B sizes 1 to n. We must show that Algorithm 6 returns the correct result for A sizes 1 to n and B sizes 1 to $n+1$. Suppose our array size is $n > 1$. Then we skip lines (1-7), as we fail to meet its conditional statement ($n = 1$), and so we start at line 8. We consider the midpoint of A, which we take to be $A[\frac{n}{2}-1]$ (the -1 accounts for starting at 0 rather than 1). Because A and B are both sorted arrays, and are identical up to some removed element e, if $A[\frac{n}{2}-1] > B[\frac{n}{2}-1]$, then that means that an element was removed from A from $A[0]$ to $A[\frac{n}{2}-1]$, inclusive, and each element after that removed element effectively shifted left. Thus, our element is contained in the subarray starting at $A[0]$ with $\frac{n}{2}$ elements. In that case, we can call the algorithm to return the missing element on A up until $\frac{n}{2}$ elements and B up until $\frac{n}{2}+1$ elements (as the size of B is always one more than A). By our precondition, these are valid arrays, and by our inductive hypothesis, this will yield the correct result, as $\frac{n}{2} \leq (n-1)$ for $n > 1$. Now, in the other case $A[\frac{n}{2}-1] \leq B[\frac{n}{2}-1]$. In this case, our lists must be indentical at least up until the $(\frac{n}{2}-1)$th position. Thus, our missing element must be in the subarray from the $\frac{n}{2}$th position to the nth position onwards (e.g. with $\frac{n}{2}$ elements). Thus, we can call the algorithm to return the missing element on $A+\frac{n}{2}$ with $\frac{n}{2}$ elements and $B+\frac{n}{2}$ with $\frac{n}{2}+1$ elements. By our precondition, these are valid arrays, and by our inductive hypothesis, this will yield the correct result, as $\frac{n}{2} \leq (n-1)$ for $n > 1$. Thus, we conclude that our algorithm is correct and yields the missing element from A.

Complexity Analysis:
In each recursive call of the algorithm, the algorithm makes constant comparisons ($O(1)$) and returns the algorithm with arrays A and B cut in half, which is halved in each subsequent call, reducing the problem size until reaching its base case condition, where n is 1. Because the size of A and B is halved with each call and runs until $n = 1$, and each call itself is $O(1)$, this algorithm runs in $O(\log(n))$ time.

(b) Give the best algorithm you can for determining which number is missing if the array is not sorted, and analyze its asymptotic worstcase running time.

The algorithm below aims to solve this problem.

Algorithm 7: Find Missing Element in Array from 1 to $n+1$

```
Input  : Array $A$ of $n$ elements
Output: Missing element $e \in \{1, 2, ..., n+1\}$
Initialize integer $i = 0$
Initially B is a size $n+2$ array
Initially integer $x$ is uninitialized
foreach $i$ such that $1 \leq i \leq n+1$ do
    B[i] = 0
end
Set $i = 0$
foreach $i$ such that $0 \leq i < n$ do
    Set $x$ = A[i]
    B[x] = 1
end
foreach $i$ such that $1 \leq i \leq n+1$ do
    if B[i] is 0 then
        return $i$
    end
end
```

We now prove the correctness of our algorithm. We know that there is only one element missing in our array, A. We create another array B that has $n+2$ elements (to account for the fact that we ignore the 0th element, we start one element later), and we initialize each element to be 0. Next, we iterate through every element of A. If we find an element x in A, we set the xth position in B (B[x]) to 1 to indicate that such an element was found. Finally, we iterate through every element in B once more. We should have every element be set to 1, except for 1 element, which should be zero (the element that was not included in A and thus was not accounted for). Thus, at the first instance we find a 0 in our B array, we print out the index of the element, which by our setup necessarily must be the element we were missing. Thus, we conclude our algorithm yields the correct result.

Complexity Analysis:

In the first for each loop, for each i between 1 (inclusive) and $n+1$ (inclusive), we perform a constant operation ($O(1)$), initializing i. The loop runs $n+1$ times, so this process is $O(n)$.
In the next for each loop, for each i between 0 (inclusive) and n (exclusive), we perform a constant operation ($O(1)$), setting $x = A[i]$ and $B[x] = 1$. The loop runs n times, so this process is $O(n)$.

In the next for each loop, for each i between 0 (inclusive) and n (exclusive), we perform a constant operation ($O(1)$), setting $x = A[i]$ and $B[x] = 1$. The loop runs n times, so this process is $O(n)$.

In the final for each loop, for each i between 1 (inclusive) and $n+1$ (inclusive), we perform a constant operation ($O(1)$), checking if $B[i] = 0$. The loop runs $n+1$ times, so this process is $O(n)$.

Adding up our time complexities, we have $O(n) + O(n) + O(n) = O(n)$. Thus, our algorithm runs in $O(n)$ time.