Exercises (not part of formal assignment, not graded)

**DPV 1.3** A $d$-ary tree is a rooted tree in which each node has at most $d$ children. Show that any $d$-ary tree with $n$ nodes must have a depth of $\Omega(\log n / \log d)$. Can you give a precise formula for the minimum depth it could possibly have?

**Solution:** Any $d$-ary tree with $n$ nodes must have a depth of $\Omega(\log n / \log d)$:

A $d$-ary tree of height $h$ has at most $1 + d + \ldots + d^h = (d^{h+1} - 1)/(d - 1)$ vertices. We can prove this inductively.

(a) **Claim:** A $d$-ary tree of height $h$ has at most $1 + d + \ldots + d^h = (d^{h+1} - 1)/(d - 1)$ vertices.

(b) **Base Case:** When the height of the tree is 1, the number of vertices is at most $1 + d = (d^2 - 1)/(d - 1)$. Therefore, the statement is true when $h = 1$.

(c) **Induction Hypothesis:** The statement is true when the height $h$ of the tree is $\leq k$.

(d) **Induction Step:** We will show that the statement is still true when $h = k + 1$. A $d$-ary tree of height $k + 1$ consists of a root that has at most $d$ children, each of which is the root of a subtree with height at most $k$. By the Induction Hypothesis, the number of vertices in a $d$-ary tree of height $\leq k$ is at most $1 + d + \ldots + d^k = (d^{k+1} - 1)/(d - 1)$. Therefore, a tree with height $k + 1$ has at most $1 + d(1 + d + \ldots + d^k)$ vertices. Hence a tree with height $k + 1$ has at most $(d^{k+2} - 1)/(d - 1)$ vertices. We have completed the induction.

The height of a tree is equal to the maximum depth $D$ of any node in the tree. For a tree of height $h$, the maximum number of vertices $n \leq (d^h - 1)/(d - 1)$. Therefore,

\[
d^h + 1 \geq n \cdot (d - 1) + 1
\]

\[
(h + 1) \geq \log_d(n \cdot (d - 1) + 1)
\]

\[
D = h \geq \left(\log_d(n \cdot (d - 1) + 1) - 1\right)
\]

By the definition of Big-$\Omega$, the maximum depth $D \geq (\log_d(n \cdot (d - 1) + 1) - 1) = \Omega(\log_d(n)) = \Omega(\log (n)/\log (d))$ is true.

To obtain the specific formula:

Let $d =$ maximum number of children, $h =$ height of tree

Geometric series:

\[
\sum \text{nodes of complete tree} = \frac{1 - d^{h+1}}{1 - d}
\]

Using $\sum \text{nodes of complete tree} = n$:

\[
n(1 - d) - 1 = -d^{h+1}
\]

\[
n(d - 1) + 1 = d^{h+1}
\]

\[
\log_d(n(d - 1) + 1) = d^{h+1}
\]

\[
\frac{\log [n(d - 1) + 1]}{\log d} = h
\]

This represents the height of a graph of a complete tree. To get the height of an incomplete tree, we take the ceiling of this equation. Incomplete levels of the tree will be rounded up (eg., $h = 2.334 \rightarrow h = 3$)

\[
\left\lceil\frac{\log [n(d - 1) + 1]}{\log d}\right\rceil = h
\]
**DPV 3.1** Perform a depth-first search on the following graph; whenever there’s a choice of vertices, pick the one that is alphabetically first. Classify each edge as a tree edge or back edge, and give the pre and post number of each vertex.

Red edges are tree edges. Black edges are back edges.

**DPV 3.3** Run the DFS-based topological ordering algorithm on the following graph. Whenever you have a choice of vertices to explore, always pick the one that is alphabetically first.

**Solution:**

a) Pre and post numbers are assigned above.

b) Sources: A, B. They have the highest post numbers with no incoming edges. Sinks: G, H. They have the lowest post numbers with no outgoing edges.

c) The topological ordering found by this algorithm: B,A,C,E,D,F,H,G.

d) There are a total of 8 topological orderings. A and B can be switched, as can D and E, as well as G and H. $2 \cdot 2 \cdot 2 = 8$. 
DPV 3.4 Run the strongly connected components algorithm on the following directed graphs $G$. When doing DFS on $G^R$: whenever there is a choice of vertices to explore, always pick the one that is alphabetically first.

Reverse the graph and run the DFS on the reverse graph. Highest post number found labeled in red: Vertex C.

Run DFS on the given graph starting with the vertex C (highest post number in the reverse graph). This guarantees you are starting in a sink SCC. Note that DFS is only run until it is stuck, but will not move to a new node.

Group all vertices together to make a SCC. Start DFS again at the next highest post number that has not yet been visited.
Continue grouping and running DFS
Solution:

a) \{C,D,F,J\} → \{H,I,G\} → \{A\} → \{E\} → \{B\}

b) Source SCC = \{B\}
   Sink SCC = \{C,D,F,J\}

c) The last graph is the meta-graph for this graph.

d) Two edges need to be added: \{C,D,F,J\} to \{E\} and \{E\} to \{B\}. The edge can be added to any node in the SCC. Because both E and B have edges to \{A\} and \{H,I,G\} we only need to connect \{B\}, \{E\}, and \{C,D,F,J\} to create a strongly connected graph. We can check this by running explore() from any node, and we should be able to reach every other node.

Using the same idea as in part (a), we can construct the meta-graph as follows

Solution:

a) \{D,F,G,H,I\} → [C] → \{A,B,E\}

b) Source SCC = \{A,B,E\}
   Sink SCC = \{D,F,G,H,I\}

c) Meta-graph shown above.

d) One edge from Sink SCC to the Source SCC will make this a strongly connected graph.
DPV 3.10 Rewrite the explore() procedure (Figure 3.3) so that it is non-recursive (that is, explicitly use a stack). The calls to previsit() and postvisit() should be positioned so that they have the same effect as in the recursive procedure.

Solution:

```plaintext
explore(Graph g, Vertex v)
    visited(v) = true
    previsit(v)
    Stack s
    s.push(v)
    while(s.size() > 0)
        v' = s.top()
        flag = 0
        for each edge (v',u) in E:
            if not visited(u):
                previsit(u)
                visited(u) = true
                s.push(u)
                flag = 1
                break from for loop
        if flag == 0: // all neighbors visited
            stack.pop() // all neighbors visited
    postvisit(v')
```
Problems (formal assignment, graded)

1. Sort each group of functions in increasing order of asymptotic (Big-O) growth. If two functions have
the same asymptotic growth, indicate this.

   (a) $2^{2^{10}} n$, $1/n$, $\log n$, $n^2$, $2^{10n}$, $n(n - 1)/2$, $n\sqrt{n}$, $n \log n$

<table>
<thead>
<tr>
<th>Solution:</th>
<th>$1/n &lt; \log n &lt; 2^{2^{10}} n &lt; n \log n &lt; n\sqrt{n} &lt; n^2 = n(n - 1)/2 &lt; 2^{10n}$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$n \log n &lt; n\sqrt{n}$</td>
</tr>
<tr>
<td>Solution:</td>
<td>Show that $n \log n \in O(n\sqrt{n})$</td>
</tr>
<tr>
<td></td>
<td>$\lim_{n \to \infty} \frac{n \log n}{n\sqrt{n}} = \lim_{n \to \infty} \frac{\log n}{\sqrt{n}}$</td>
</tr>
<tr>
<td></td>
<td>(Using L’Hospital’s rule)</td>
</tr>
<tr>
<td></td>
<td>$= \lim_{n \to \infty} \frac{1/2\sqrt{n}}{1/n} = \lim_{n \to \infty} \frac{2}{n\sqrt{n}} = 0 &lt; \infty$</td>
</tr>
<tr>
<td>AND,</td>
<td>Show that $n\sqrt{n} \notin O(n \log n)$</td>
</tr>
<tr>
<td></td>
<td>$\lim_{n \to \infty} \frac{n\sqrt{n}}{n \log n} = \lim_{n \to \infty} \frac{\sqrt{n}}{\log n}$</td>
</tr>
<tr>
<td></td>
<td>(Using L’Hospital’s rule)</td>
</tr>
<tr>
<td></td>
<td>$= \lim_{n \to \infty} \frac{1/n}{1/2\sqrt{n}} = \lim_{n \to \infty} \frac{\sqrt{n}}{2} \neq \infty$</td>
</tr>
<tr>
<td></td>
<td>$n^2 = \frac{n(n-1)}{2}$</td>
</tr>
<tr>
<td>Solution:</td>
<td>Show that $n^2 \in \Theta\left(\frac{n(n-1)}{2}\right)$ ($n^2 \in O\left(\frac{n(n-1)}{2}\right)$ and $\frac{n(n-1)}{2} \in O(n^2)$)</td>
</tr>
<tr>
<td></td>
<td>To show $n^2 \in O\left(\frac{n(n-1)}{2}\right)$: take $c = 4$, $N = 2$</td>
</tr>
<tr>
<td></td>
<td>To show $\frac{n(n-1)}{2} \in O(n^2)$: take $c = 1$, $N = 1$</td>
</tr>
</tbody>
</table>
(b) \( n^{\sqrt{n}}, 2^n, n^{10}, 3^n, n^{\log n}, \left( \frac{n}{n/2} \right), \log(n^n), n! \)

**Solution:** \( \log(n^n) < n^{10} < n^{\log n} < n^{\sqrt{n}} < \left( \frac{n}{n/2} \right) < 2^n < 3^n < n! \)

- \( n^{\sqrt{n}} < 2^n \)
  
  Hint: \( n^{\sqrt{n}} \) can be written as \( 2^{\log n \cdot \sqrt{n}} \).

- \( 3^n < n! \)
  
  Solution: Show that \( 3^n \in \mathcal{O}(n!) \):

  \[
  \lim_{n \to \infty} \frac{3^n}{n!} = \lim_{n \to \infty} \frac{3 \cdot 3 \cdot 3 \cdots (n \text{ times})}{n \cdot (n-1) \cdots 1} 
  = \lim_{n \to \infty} \frac{3}{n} \cdot \frac{3}{n-1} \cdots \frac{3}{2} \cdot 1
  \leq \lim_{n \to \infty} \frac{3 \cdot 3 \cdot 3}{3 \cdot 3 \cdot 2 \cdot 1}
  = \lim_{n \to \infty} \frac{9}{2} = 0 < \infty
  
  \]
  
  AND, show that \( n! \notin \mathcal{O}(3^n) \) by showing that

  \[
  \lim_{n \to \infty} \frac{n!}{3^n} < \infty
  \]

- \( \left( \frac{n}{n/2} \right) < 2^n \)
  
  Solution: Show that \( \left( \frac{n}{n/2} \right) \in \mathcal{O}(2^n) \)

  Using Stirling’s approximation as \( n \to \infty \),

  \[
  \left( \frac{n}{n/2} \right) = \frac{n!}{(n/2)!(n/2)!} \approx \frac{\sqrt{2\pi n} \left( \frac{n}{e} \right)^n}{\left( \sqrt{\pi n} \left( \frac{n}{2e} \right)^{n/2} \right)^2} \approx \frac{\sqrt{2} \cdot 2^n}{\sqrt{\pi n}}
  
  \]
  
  Therefore,

  \[
  \lim_{n \to \infty} \frac{\left( \frac{n}{n/2} \right)}{2^n} \approx \lim_{n \to \infty} \frac{\sqrt{2} \cdot 2^n}{2 \cdot 2^n} = \lim_{n \to \infty} \frac{\sqrt{2}}{\sqrt{\pi n}} = 0
  
  \]
  
  Similarly, show that \( 2^n \notin \mathcal{O}\left( \frac{n}{n/2} \right) \)}
2. (a) Find (with proof) a function \( f_1 \) mapping positive integers to positive integers such that \( f_1(2n) \) is \( \mathcal{O}(f_1(n)) \).

**Solution:** Consider \( f_1(n) = n \).

Let \( g(n) = f_1(n) = n \) and \( h(n) = f_1(2n) = 2n \).

We now have to prove that \( h(n) \in \mathcal{O}(g(n)) \):

By the definition of Big-O, \( h(n) \in \mathcal{O}(g(n)) \) if \( \exists \) positive constants \( c, N \) such that \( h(n) \leq c \cdot g(n), \forall n > N \)

Since, \( 2n \leq 2 \cdot n, \forall n > 0 \) (here \( c = 2, N = 0 \)), \( h(n) \in \mathcal{O}(g(n)) \)

(b) Find (with proof) a function \( f_2 \) mapping positive integers to positive integers such that \( f_2(2n) \) is not \( \mathcal{O}(f_2(n)) \).

**Solution:** Consider \( f_2(n) = 2^n \).

Let \( g(n) = f_2(n) = 2^n \) and \( h(n) = f_2(2n) = 2^{2n} = (2^2)^n = 4^n \).

We now have to prove that \( h(n) \notin \mathcal{O}(g(n)) \):

Assume toward a contradiction that \( h(n) \in \mathcal{O}(g(n)) \). By the definition of Big-O, this means that there exist constants \( c > 0, N \) such that \( h(n) \leq c \cdot g(n), \forall n \geq N \). But this is equivalent to saying \( 2^n \cdot 2^n \leq c \cdot 2^n \forall n \geq N \implies 2^n \leq c \forall n \geq N \). But the function \( 2^n \) is monotonically increasing in \( n > 0 \), and cannot be bounded by the assumed constant \( c \) for all \( n > N \).

Alternate proof using limits:

By the definition of Big-O, \( h(n) \in \mathcal{O}(g(n)) \) if \( \lim_{n \to \infty} \frac{h(n)}{g(n)} < \infty \)

\[
\lim_{n \to \infty} \frac{h(n)}{g(n)} = \lim_{n \to \infty} \frac{4^n}{2^n} = \lim_{n \to \infty} 2^n \neq \infty
\]

(c) Prove that if \( f(n) \) is \( \mathcal{O}(g(n)) \), and \( g(n) \) is \( \mathcal{O}(h(n)) \), then \( f(n) \) is \( \mathcal{O}(h(n)) \).

**Solution:** By the definition of Big-O, there exist positive integers \( N_1 \) and \( N_2 \) and positive constants \( c_1 \) and \( c_2 \) such that \( f(n) \leq c_1 \cdot g(n) \) for \( n > N_1 \) and \( g(n) \leq c_2 \cdot h(n) \) for \( n > N_2 \).

Let \( N_0 = \max(N_1, N_2) \) and \( c_0 = c_1c_2 \).

\[
f(n) \leq c_1 \cdot g(n), \forall n > N_0 \tag{1}
g(n) \leq c_2 \cdot h(n), \forall n > N_0 \tag{2}
\]

(1) and (2) imply,

\[
f(n) \leq c_1 \cdot g(n) \leq c_1c_2 \cdot h(n), \forall n > N_0
\]

Thus, \( f(n) \) is \( \mathcal{O}(h(n)) \)
(d) Prove or disprove: if \( f \) is not \( \mathcal{O}(g) \), then \( g \) is \( \mathcal{O}(f) \).

**Solution:**
A counterexample can disprove the above claim:

Let \( f(n) = 2^n \cdot \sin n \), \( g(n) = n \). We will show that \( f(n) \notin \mathcal{O}(g(n)) \) and \( g(n) \notin \mathcal{O}(f(n)) \)

**Definition:** \( f(n) \notin \mathcal{O}(g(n)) \) if for all choices of positive \( c \) and \( N \), there exists some \( n > N \) such that \( f(n) > cg(n) \).

Showing that \( 2^n \cdot \sin n \notin \mathcal{O}(n) \):
Choose any \( c, N > 0 \). We can choose \( n > N \) such that:

i. \( \sin n = d > 0 \)

ii. \( 2^d n > cn \)

Similarly, show that \( n \notin \mathcal{O}(2^n \cdot \sin n) \):
Choose any \( c, N > 0 \). We can choose \( n > N \) such that:

i. \( \sin n = e < 0 \)

ii. \( cn > 2^e n \)
3. (DPV 3.22) Give an efficient algorithm which takes as input a directed graph $G = (V,E)$ and determines whether or not there is a vertex $s \in V$ from which all other vertices are reachable.

**Solution:** Let us call a vertex from which all other vertices are reachable, a “vista vertex”. If the graph has a vista vertex, then it must have only one source SCC (since two source SCC’s are not reachable from each other), which must contain the vista vertex (if it is in any other SCC, there is no path from the vista vertex to the source SCC). (Recall the property that the meta-graph of a digraph’s SCCs is a DAG and therefore will have a source SCC.) Moreover, in this case, every vertex in the source SCC will be a vista vertex. Our aim now is to find a vertex in a source SCC. We run DFS starting from any vertex and mark the vertex with the highest post value. This must be in a source SCC. We now run **explore** from this vertex to check if we can reach all vertices.

Pseudocode:

```
procedure vista_vertex(G)
    pick a vertex u
    DFS(G,u)
    Find vertex with highest post number (call it v)
    explore(G,v)
    if all vertices visited:
        return v
    else:
        return no_vista_vertex_found
```

Since the algorithm just uses decomposition into SCCs and DFS which are linear, the running time is linear.
4. (DPV 3.25) You are given a directed graph in which each node \( u \in V \) has an associated price \( p_u \) which is a positive integer. Define the array \( \text{cost} \) as follows: for each \( u \in V \)

\[
\text{cost}[u] = \text{price of the cheapest node reachable}
\]

For instance, in the graph below (with prices shown for each vertex), the \( \text{cost} \) values of the nodes \( A, B, C, D, E, F \) are 2, 1, 4, 1, 4, 5 respectively.

Your goal is to design an algorithm that fills in the entire \( \text{cost} \) array (i.e., for all vertices).

(a) Give a linear-time algorithm that works for directed acyclic graphs. (Hint: Handle the vertices in a particular order.)

**Solution:** Consider the vertices of the DAG in topological order. Let \( v_1, \ldots, v_n \) be the linearized order. Vertices reachable from any vertex (let’s say \( v_k \)) will be among the vertices which are “after” (smaller post number) \( v_k \) in the linearized order. Once we have updated \( \text{costs} \) for vertices \( v_{k+1}, \ldots, v_n \), \( \text{cost} \) of \( v_k \) will be minimum of \( \text{cost} \) of vertices connected to \( v_k \) (including itself). This is because any path from vertex \( v_k \) will be through its adjacent vertices; \( \text{costs} \) for which are already calculated. We implement an algorithm which linearizes the DAG and calculates \( \text{costs} \) in reverse topological order:

**Pseudocode:**

```plaintext
procedure cheapest_cost(G)
    DFS(G,u) // where u is a randomly picked vertex
    (v1, v2, ..., vn) ← decreasing order of post[vi]
    For i = n to 1:
        cost[vi] = price(vi)
        for all (vi, vj) ∈ E
            if cost[vj] < cost[vi]
                cost[vi] = cost[vj]

    The time for linearizing a DAG is linear. To find minimum cost, we visit each edge at most once and hence the time is linear.
```
(b) Extend this to a linear-time algorithm that works for all directed graphs. (Hint: Recall the “two-tiered” structure of directed graphs.)

Solution: For a general directed graph, the cost of any two nodes in the same strongly connected component will be same since both are reachable from each other. Hence, it is sufficient to run the above algorithm on the DAG of the strongly connected components of the graph. For a node corresponding to strongly connected component $C$, we take $price_C = \min(price_u)$ for all $u$ in $C$. Costs of all $u$ in $C$ will be $price_C$.

Once we have found SCCs, meta-nodes can be topologically sorted by arranging them in decreasing order of their highest post numbers.

Pseudocode:

```
procedure cheapest_cost(G)
    DFS($G^R$, u) // where u is a randomly picked vertex
    Run undirected connected components algorithm on G (DPV Section 3.4.2),
    processing vertices in decreasing order of post numbers
    post[C] = max(post(u)) for all u in C
    For all c in C (where C is the set of all SCC’s):
        price[c] = min(price(u)) for all u in c
        (c1, c2, ..., cn) ← decreasing order of post(ci)
    For i = n to 1:
        cost[ci] = price[ci]
        for all (ci, cj) ∈ E
            if cost[cj] < cost[ci]
                cost[ci] = cost[cj]
```
5. A binary tree is a rooted tree in which each node has at most two children. Show that in any binary tree the number of nodes with two children is exactly one less than the number of leaves. (Hint: induction.)

**Solution:** We will prove this by induction on number of nodes in a binary tree $T$.

Let $n_0(T) = \text{number of leaves of binary tree } T$, $n_2(T) = \text{number of nodes in } T \text{ with 2 children}$

(a) Claim: A binary tree $T$ with $n$ vertices satisfies $n_0(T) - 1 = n_2(T)$

(b) Base Case: A binary tree with 1 vertex has 1 leaf and no vertices with 2 children. Hence, the condition is satisfied.

(c) Induction Hypothesis: A binary tree $T'$ with $n$ vertices satisfies $n_0(T') - 1 = n_2(T')$.

(d) Induction Step: We will show the condition to be true for a binary tree $T$ with $n + 1$ vertices.

Let $T$ be an arbitrary binary tree with $n + 1$ vertices. Let $v$ be a leaf of the tree. Since the tree has more than one vertex, $v$ is not the root and it therefore has a parent $u$. Let $T'$ be obtained by deleting the leaf $v$.

Case 1: If the parent $u$ had 2 children.

- Number of leaves in the tree reduces by 1, $n_0(T') = n_0(T) - 1$
- Number of vertices with 2 children reduces by 1, $n_2(T') = n_2(T) - 1$
- By induction hypothesis, we know $n_0(T') = n_2(T') + 1$
- $n_0(T') = n_2(T') + 1 \Rightarrow n_0(T) = n_2(T) + 1$
- Therefore, the condition holds true for $T$ with $n + 1$ vertices.

Case 2: If the parent $u$ had 1 child. Now $u$ becomes a leaf. Number of leaves and vertices with 2 children remain unchanged:

- $n_0(T') = n_0(T)$
- $n_2(T') = n_2(T)$
- $n_0(T') = n_2(T') + 1 \Rightarrow n_0(T) = n_2(T) + 1$
- Therefore, the condition holds true for $T$ with $n + 1$ vertices.

Hence proved by induction.
6. In this problem the input includes an array \( A \) such that \( A[0 \ldots n-1] \) contains \( n \) integers that are sorted into non-decreasing order: \( A[i] \leq A[i+1] \) for \( i = 0,1,\ldots,n-2 \). The array \( A \) may contain repeated elements, e.g. \( A = [0,1,1,2,3,3,3,4,5,5,6,6,6] \)

(a) Describe carefully an algorithm \( \text{COUNT}(A,x) \) that, given an array \( A \), and an integer \( x \), returns the number of occurrences of \( x \) in the array \( A \). Your algorithm should be similar to binary search, and must run in \( O(\log n) \).

**Solution:** First we will implement the function \( \text{FIRST\_INDEX}(A,x,\text{low, high}) \) which returns the smallest \( i \) such that \( \text{low} \leq i < \text{high} \) and \( A[i] \geq x \). This is done by tweaking the binary search algorithm so that it continues searching for the first index of the “key” even if it has already found one occurrence. If no such index exists, then the function returns \( \text{high} \).

We implement \( \text{FIRST\_INDEX} \) as follows:

\[
\text{FIRST\_INDEX}(A, x, \text{low, high})
\]

\[
\text{if low = high }
\]

\[
\text{return low}
\]

\[
\text{mid = (low + high)/2}
\]

\[
\text{if A[mid] < x}
\]

\[
\text{then return FIRST\_INDEX(A, x, mid + 1, high)}
\]

\[
\text{if A[mid] \geq x}
\]

\[
\text{then return FIRST\_INDEX(A, x, low, mid)}
\]

Then we implement \( \text{COUNT}(A,x) \) which will give the number of occurrences of \( x \) in array \( A \). It does this as shown below:

\[
\text{COUNT}(A,x)
\]

\[
\text{return FIRST\_INDEX(A, x+1, 0, n) - FIRST\_INDEX(A, x, 0, n)}.
\]

**Running time:**

There are two calls to \( \text{FIRST\_INDEX} \) by \( \text{COUNT} \). Therefore, running time of our algorithm is the running time of \( \text{FIRST\_INDEX} \).

Let \( n = \text{high} - \text{low} \). If \( n > 1 \), then \( \text{FIRST\_INDEX} \) makes a recursive call to either the left half or the right half of the array i.e. single subproblem of size \( n/2 \). This gives the recursive relation

\[
T(n) = T(n/2) + \Theta(1)
\]

which can be solved using the Master Theorem to give \( T(n) = \Theta(\log n) \).
(b) Prove that your algorithm always terminates.

**Solution:** We need only show that FIRST_INDEX terminates. The algorithm terminates because every recursive call to FIRST_INDEX processes a smaller range of $A$ (the difference $\text{high} - \text{low}$ strictly decreases), so every call will eventually terminate at the base case where $\text{low} = \text{high}$.

**Proof by induction:**

(a) **Claim:** The algorithm FIRST_INDEX terminates with any input of size $n$, $n \geq 0$

(b) **Base Case:** The algorithm terminates with input of size 0: since $\text{low} = \text{high}$ it terminates

(c) **Induction Hypothesis:** The algorithm FIRST_INDEX terminates with any input of size $n$, $\forall n \leq k$, $k \geq 0$

(d) **Induction Step:** We need to show that FIRST_INDEX terminates with input of size $k + 1$

   The algorithm makes a recursive call on either the left half of the array or the right half of the array. The input sizes on both these recursive calls is $< k + 1$. By induction hypothesis we know that the algorithm terminates for input sizes $\leq k$. Hence the algorithm will terminate.

This completes the proof

(c) Prove that when your algorithm terminates, it terminates with the correct answer.

**Solution:** The function FIRST_INDEX terminates with the correct answer because if $A[mid]$ is less than $x$, then no index less than $mid$ can contain any element that is at least $x$, so we can recurse to the right half. On the other hand, if $A[mid]$ is at least $x$, then the smallest index $i$ such that $A[i]$ is at least $x$ must be at most equal to $mid$, so we can recurse to the left half.

Finally, the difference between $\text{FIRST_INDEX}(A, x+1, 0, n) - \text{FIRST_INDEX}(A, x, 0, n)$ must be the number of indices that contain exactly $x$. 

7. You are given a vertex-weighted graph. Consider the following definitions.

Independent set: a set of vertices in a graph, no two of which are adjacent.
Weight of independent set: sum of weights of vertices in the set.
Max-weight independent set problem: Find an independent set which has maximum weight.

Consider the following “greedy” approach to finding a max-weight independent set in a given vertex-weighted graph:

1) Start with an empty set \( X \).
2) For each vertex \( v \) of the graph, in decreasing order of weight:
   - add vertex \( v \) to the set \( X \) if \( v \) is not adjacent to any vertex in \( X \).
3) Return \( X, \text{weight}(X) \).

(a) Show that the “greedy” approach for the max-weight independent set is not optimal, by exhibiting a small counterexample.

\[ \text{Solution:} \text{ Consider the graph shown below with } w(B) = 3, w(A) = w(C) = 2: } \]

\[ \begin{array}{ccc}
2 & 3 & 2 \\
A & B & C \\
\end{array} \]

The greedy approach adds \( B \) to set \( X \) and is unable to add more vertices. It returns \( \{B\} \) with weight 3. However the max-weight independent set for the above is \( \{A, C\} \) with weight \( = 4 \).

(b) How badly suboptimal can greed be, relative to optimal? Please clearly explain your definition of “badly suboptimal”.

\[ \text{Solution:} \text{ We define the suboptimality of greed relative to optimal as the ratio of the weight of the optimum max-weight independent set to the weight of the greedy max-weight independent set. The larger the ratio, the greater the suboptimality of the greedy approach. } \]

Consider the \( n \)-vertex “star” below:

\[ \begin{array}{c}
\text{Worst case behavior of greedy approach:}
\hline
\bullet \text{ the addition of first vertex } u \text{ to } X \text{ forbids other vertices from getting added (all other vertices are adjacent to } u) \\
\bullet \text{ Weight of } u \text{ is just greater than that of the other vertices by a small amount } (\epsilon) \\
\text{Therefore, suboptimality } = \frac{n - 1}{1 + \epsilon} \in \Omega(n). \\
\end{array} \]