These "model solutions" are provided to give an idea of the level of detail and formality that would receive full credit in homework writeups or exams.

1 Algorithm design.

Give a logarithmic time algorithm that, given two integers \(a\) and \(n\), where \(n\) is a nonnegative power of 2, calculates \(a^n\).

Solution

Algorithm. Our algorithm will recursively calculate \(a^n = (a^{2^p})^2\) until \(n = 1\) where \(a^1 = a\).

Input: two integers \(a\) and \(n\) such that \(n\) is a positive power of 2

Output: \(a^n\)

procedure pow(a, n):
    if \(n == 1\):
        return \(a\)
    else:
        \(b = \text{pow}(a, n/2)\)
        return \(b \cdot b\)

Running time Let \(T(n)\) be the number of multiplications. At each step \(n\) is divided by 2 and we do one multiplication, so \(T(n) = T(\frac{n}{2}) + 1\). By the Master Theorem, \(T(n) = O(\log n)\).

Proof of correctness. Since \(n\) is a power of 2, we can rewrite it as \(n = 2^p\), where \(p \geq 0\). We now prove by induction on \(p\) that \(\text{pow}(a, 2^p) = a^{2^p}\).

- Base case: Our base case is \(p = 0\). We have \(n = 2^0 = 1\) and \(\text{pow}(a, 1) = a = a^1\).

- Induction Hypothesis: Let’s assume that \(\text{pow}(a, 2^p) = a^{2^p}\) for some \(p \geq 0\).

- Induction Step: We now show that \(\text{pow}(a, 2^{p+1}) = a^{2^{p+1}}\). From our algorithm, \(\text{pow}(a, 2^{p+1}) = b^2\) where \(b = \text{pow}(a, 2^p)\). From our induction hypothesis we know that \(\text{pow}(a, 2^p) = a^{2^p}\). Thus \(b = a^{2^p}\) and \(\text{pow}(a, 2^{p+1}) = b^2 = (a^{2^p})^2 = a^{2^{p+1}}\).

Finally, by induction, we have proven that \(\text{pow}(a, 2^p) = a^{2^p}\) for all \(p \geq 0\).

2 Proofs and counterexamples.

Exercise 3.6.

In an undirected graph, the degree \(d(u)\) of a vertex \(u\) is the number of neighbors \(u\) has, or equivalently, the number of edges incident upon it. In a directed graph, we distinguish between the indegree \(d_{in}(u)\), which is the number of edges into \(u\), and the outdegree \(d_{out}(u)\), the number of edges leaving \(u\).

(a) Show that in an undirected graph, \(\sum_{u \in V} d(u) = 2|E|\).
(b) Use part (a) to show that in an undirected graph, there must be an even number of vertices whose degree is odd.

(c) Does a similar statement hold for the number of vertices with odd indegree in a directed graph?

Solution

(a) Direct proof. For a vertex \( u \), \( d(u) \) is the number of edges incident upon it. So \( \sum_{u \in V} d(u) \) is the count of all edges incident on all the vertices. As an edge has exactly two vertices, all edges are counted exactly twice. So we have \( \sum_{u \in V} d(u) = 2|E| \).

(b) Proof by contradiction. Suppose that there are an odd number of vertices with an odd degree. Then the sum of all degrees is odd. Yet, we proved in (a) that the sum of all degrees is even (as \( 2|E| \) must be an even number). This contradicts our assumption. So there must be an even number of vertices with an odd degree.

(c) Proof by counterexample. This statement does not hold in a directed graph when considering indegrees. For example, in the 2 vertex simple graph \((A \rightarrow B)\), we have \( d_{in}(A) = 0 \) and \( d_{in}(B) = 1 \). So, we have only 1 (odd) vertex with an odd degree.

3 Proof by induction

Example 1

Prove that it is possible to color the regions formed by any number of lines in the plane using only two colors.

Solution

We will prove the above statement by weak induction.

Base case. Let \( n \) be the number of lines in the plane. Clearly the statement is true when \( n = 1 \). We use one color for left (or top) side of the line, and the other color for the right (or bottom) side of the line.

Inductive hypothesis. Now, assume as our inductive hypothesis that we can color the regions formed by \( n \) lines in the plane where \( n > 1 \) with only two colors.

Inductive step. Now, consider the case when we have \( n + 1 \) lines. First, we remove one line and note that by our inductive hypothesis we can color all regions with only two colors. Next, we add back the removed line. On only one side of the line, we swap all the colors. We now have \( n + 1 \) lines in the plane and a possible two-coloring since either:

- A region is split by the newly added line in which case we flipped the color for only half of the region.
- Or a region is not split by the line in which case we may have flipped the color of the region or kept it the same. Since all neighboring regions on the same side of the line were also flipped or kept the same this change does affect the correctness of the coloring.

So, by induction we have show that \( \forall n > 0 \) it is possible to color the regions formed by any number of lines in the plane using only two colors.
Note: Weak induction

As our inductive hypothesis was only concerned with using a value of $n$ to show the claim holds for a value of $n + 1$ we used weak induction. A proof by strong induction would have required us to assume the claim true for any number of lines $k$ such that $1 < k < n$ in order to show the statement holds for $n$ lines.

Example 2

Prove that if $n$ is a natural number and $1 + x > 0$, then $(1 + x)^n \geq 1 + nx$.

Solution

We will prove the claim by induction on $n$.

Base case. When $n = 0$ the claim holds since $(1 + x)^0 \geq 1 + 0$.

Inductive hypothesis. Now, assume as our inductive hypothesis that $(1 + x)^n \geq 1 + nx$ for some value of $n > 0$.

Inductive step. Now, can show the following chain of inequalities:

\[
(1 + x)^{n+1} = (1 + x)^n(1 + x) \\
\geq (1 + nx)(1 + x) \quad \text{by our inductive hypothesis} \\
\geq 1 + nx + x + nx^2 \\
\geq 1 + (n + 1)x + nx^2 \\
\geq 1 + (n + 1)x + nx^2 \\
\geq 1 + (n + 1)x \quad \text{since } nx^2 \geq 0 \text{ since } n \geq 0
\]

So, by induction, we have shown that $\forall n \in \mathbb{N}, (1 + x)^n \geq 1 + nx$.

4 More Examples
(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 $2^{\log n\sqrt{n}}$.

- $3^n < n!$
  
  Solution: Show that $3^n \in O(n!)$:

\[
\lim_{n \to \infty} \frac{3^n}{n!} = \lim_{n \to \infty} \frac{3 \cdot 3 \cdot 3 \cdots (n \text{ times})}{n \cdot (n-1) \cdot \cdots \cdot (n - (n-1))} = \lim_{n \to \infty} \frac{3}{n} \cdot \frac{3}{(n-1)} \cdot \cdots \cdot \frac{3}{3} \cdot \frac{3}{2} \cdot \frac{3}{1} \leq \frac{3}{n} \cdot \frac{9}{2} = 0 < \infty
\]

AND, show that $n! \notin O(3^n)$ by showing that

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

- $\left(\frac{n}{n/2}\right) < 2^n$

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

Using Stirling’s approximation as $n \to \infty$,

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

Therefore,

\[
\lim_{n \to \infty} \frac{\left(\frac{n}{n/2}\right)}{2^n} \approx \lim_{n \to \infty} \frac{\sqrt{2} \cdot 2^n}{\sqrt{\pi n} \cdot 2^n} = \lim_{n \to \infty} \frac{\sqrt{2}}{\sqrt{\pi n}} = 0
\]

Similarly, show that $2^n \notin O\left(\frac{n}{n/2}\right)$.
2. (a) Find (with proof) a function \( f_1 \) mapping positive integers to positive integers such that \( f_1(2n) \) is \( O(f_1(n)) \).

**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 \( h(n) \in O(g(n)) \):
By the definition of Big-O, \( h(n) \in O(g(n)) \) 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 \( c = 2, N = 0 \)), \( h(n) \in O(g(n)) \)

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

**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 \( h(n) \notin O(g(n)) \):
Assume toward a contradiction that \( h(n) \in O(g(n)) \). 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, \( h(n) \in O(g(n)) \) if \( \lim_{n \to \infty} \frac{h(n)}{g(n)} < \infty \)
\[
\lim_{n \to \infty} \frac{h(n)}{g(n)} = \lim_{n \to \infty} \frac{4^n}{2^n} = \lim_{n \to \infty} 2^n \Rightarrow \infty
\]

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

**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 \( O(h(n)) \)
(d) Prove or disprove: if $f$ is not $O(g)$, then $g$ is $O(f)$.

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

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

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

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

i. $\sin n = d > 0$
ii. $2^dn > cn$

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

i. $\sin n = e < 0$
ii. $cn > 2^en$
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:

```plaintext
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.
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 \( n_0(T) = \) number of leaves of binary tree \( T \), \( n_2(T) = \) number of nodes in \( T \) 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 \( n_0(T') = n_2(T') + 1 \)

\[ n_0(T') = n_2(T') + 1 \implies n_0(T) = n_2(T) + 1 \]

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:

\[ n_0(T') = n_0(T) \]
\[ n_2(T') = n_2(T) \]

\[ n_0(T') = n_2(T') + 1 \implies n_0(T) = n_2(T) + 1 \]

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 $COUNT(A, x)$ 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 $FIRST\_INDEX(A, x, low, high)$ which returns the smallest $i$ such that $low \leq i < high$ 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 $high$.

We implement $FIRST\_INDEX$ as follows:

```plaintext
FIRST\_INDEX(A, x, low, high)
    if low = high
        return low
    mid = (low + high)/2
    if A[mid] < x
        return FIRST\_INDEX(A, x, mid + 1, high)
    if A[mid] \geq x
        return FIRST\_INDEX(A, x, low, mid)
```

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

```plaintext
COUNT(A, x)
    return FIRST\_INDEX(A, x+1, 0, n) - FIRST\_INDEX(A, x, 0, n).
```

**Running time:**
There are two calls to $FIRST\_INDEX$ by $COUNT$. Therefore, running time of our algorithm is the running time of $FIRST\_INDEX$.

Let $n = high - low$. If $n > 1$, then $FIRST\_INDEX$ 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 high−low strictly decreases), so every call will eventually terminate at the base case where low = high.

Proof by induction:

(a) Claim: The algorithm FIRST_INDEX terminates with any input of size n, n ≥ 0

(b) Base Case: The algorithm terminates with input of size 0: since low = high it terminates

(c) Induction Hypothesis: The algorithm FIRST_INDEX terminates with any input of size n, ∀n ≤ k, k ≥ 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 ≤ 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 FIRST_INDEX(A, x+1, 0, n) - 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.

**Solution:** Consider the graph shown below with \( w(B) = 3 \), \( w(A) = w(C) = 2 \):

![Graph with vertices A, B, C and weights]

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”.

**Solution:** 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:

![Star graph with vertices v_1, v_2, ..., v_n]

Worst case behavior of greedy approach:

- the addition of first vertex \( u \) to \( X \) forbids other vertices from getting added (all other vertices are adjacent to \( u \))
- Weight of \( u \) is just greater than that of the other vertices by a small amount \( (\varepsilon) \)

Therefore, suboptimality = \( \frac{n-1}{1+\varepsilon} \in \Omega(n) \).