1. Describe and analyze an efficient algorithm to determine whether a given rolling die maze is solvable. Your input is a two-dimensional array \( Label[1..n, 1..n] \), where each entry \( Label[i, j] \) stores the label of the square in the \( i \)th row and \( j \)th column, where the label 0 means the square is free, and the label −1 means the square is blocked.

**Solution:** We solve this problem by reducing it to a graph reachability problem. Given the input array \( Label[1..n, 1..n] \), we construct an undirected graph \( G = (V, E) \) as follows.

- **\( V \) contains one vertex for every legal placement of a die on the board.** More explicitly, each vertex is an integer tuple \((i, j, t, f)\)—representing the die resting on row \( i \) and column \( j \) with \( t \) pips on the top face and \( f \) pips on the front face—satisfying the following constraints:
  - \( 1 \leq i \leq n \) and \( 1 \leq j \leq n \) — Square \((i, j)\) is actually on the board.
  - \( 1 \leq t \leq 6 \) and \( 1 \leq f \leq 6 \) — The die has faces with \( t \) and \( f \) pips.
  - \( f \neq t \) and \( f + t \neq 7 \) — Faces \( t \) and \( f \) are neither identical nor opposites.
  - Either \( Label[i, j] = 0 \) or \( Label[i, j] = t \) — The placement is legal.

There are at most \( 24n^2 = O(n^2) \) vertices.

- **\( E \) contains one edge for every legal move.** (The edges are undirected because the reversal of any legal move is also a legal move.) Specifically, each vertex \((i, j, t, b)\) has at most four neighbors

  \[
  (i + 1, j, l, f), \quad (i - 1, j, r, f), \quad (i, j + 1, f, b), \quad (i - 1, j, k, t),
  \]

  where \( l, r, b, \) and \( k \) are respectively the number of pips on the left, right, bottom, and back sides of the die when there are \( t \) pips on top and \( f \) pips in front. Of course each of these tuples is a neighbor if and only if represents a legal placement of the die! Since there are at most four legal moves from any legal placement of the die, each vertex has degree at most 4, so there are at most \( 48n^2 = O(n^2) \) edges altogether.

- By construction, every walk in \( G \) represents a sequence of legal moves from one legal placement to another, and every sequence of legal moves is represented by a walk in \( G \). Thus, the maze is solvable if and only if \( G \) contains a walk from some starting vertex \((1, 1, t, f)\) to some goal vertex \((n, n, t', f')\).

- We can construct \( G \) from the input array by brute force in \( O(n^2) \) time.

Once we have constructed \( G \), we find all goal vertices \((n, n, t', f')\) reachable from each starting vertex \((1, 1, t, f)\) in \( O(V + E) + O(n^2) \) time using whatever-first search. Since there are at most \( 24 = O(1) \) starting vertices, our algorithm runs in \( O(n^2) \) time overall.  

**Rubric:** This question is intended to test your ability to reduce to a standard problem an apply a standard algorithm as a black box. The gray text is not required for full credit.
2. Describe and analyze fast algorithms for the following problems. The input for each problem is an unsorted array \( A[1..n] \) of \( n \) numbers.

(a) Are there two distinct indices \( i < j \) such that \( A[i] + A[j] = 0 \)?

**Solution:** The following algorithm runs in \( O(n \log n) \) time:

\[
\text{2Sum}(A[1..n]):
\]
\[
\text{sort } A
\]
\[
i \leftarrow 1; \ \ j \leftarrow n
\]
\[
\text{while } i < j
\]
\[
\text{if } A[i] + A[j] < 0
\]
\[
i \leftarrow i + 1
\]
\[
\text{else if } A[i] + A[j] > 0
\]
\[
j \leftarrow j - 1
\]
\[
\text{else}
\]
\[
\text{return True}
\]
\[
\text{return False}
\]

**Solution:** The following algorithm runs in \( O(n) \) expected time. In any solution to \( A[i] + A[j] = 0 \), either both array entries are zero, or one is positive and the other is negative. The algorithm handles these two cases separately.

\[
\text{2Sum}(A[1..n]):
\]
\[
\text{seenZero} \leftarrow \text{False}
\]
\[
\text{for } i \leftarrow 1 \text{ to } n
\]
\[
\text{if } A[i] = 0 \text{ and seenZero}
\]
\[
\text{return True}
\]
\[
\text{else if } A[i] = 0
\]
\[
\text{seenZero} \leftarrow \text{True}
\]
\[
H \leftarrow \text{new hash table}
\]
\[
\text{for } i \leftarrow 1 \text{ to } n
\]
\[
\text{if } A[i] > 0
\]
\[
\text{INSERT}(H, A[i])
\]
\[
\text{for } i \leftarrow 1 \text{ to } n
\]
\[
\text{if } \text{CONTAINS}(H, -A[i])
\]
\[
\text{return True}
\]
\[
\text{return False}
\]

**Rubric:** This question is intended to test your ability to describe an algorithm clearly and precisely. Clear and unambiguous English is instead of pseudocode is fine, but executable C/Java/Python code is not. Full credit requires handling input arrays with no zeros, with one zero, and with multiple zeros correctly. Full credit for the first solution requires the initial sort. Full credit for the second algorithm requires the word “expected” in the time analysis; hashing is inherently randomized!
(b) Are there three distinct indices \( i < j < k \) such that \( A[i] + A[j] + A[k] = 0 \)?

**Solution:** The following algorithm runs in \( O(n^2) \) time:

```plaintext
3Sum(A[1..n]):
    sort A
    for j ← 2 to n - 1
        i ← 1; k ← n
        while i < j and j < k
                i ← i + 1
                k ← k - 1
            else
                return True
        return False
```

**Solution:** The following algorithm runs in \( O(n^2) \) expected time. In any solution to \( A[i] + A[j] + A[k] = 0 \), either all three array entries are zero, or one is positive and the other two non-positive, or one is negative and the other two non-negative. The algorithm handles each of these three cases separately.

```plaintext
3Sum(A[1..n]):
    zeros ← 0
    for i ← 1 to n
        if A[i] = 0 and zeros ≥ 2
            return True
        else if A[i] = 0
            zeros ← zeros + 1
    H⁺ ← new hash table
    H⁻ ← new hash table
    for i ← 1 to n - 1
        for j ← i + 1 to n
            if A[i] ≥ 0 and A[j] ≥ 0
                INSERT(H⁺, A[i] + A[j])
            if A[i] ≤ 0 and A[j] ≤ 0
                INSERT(H⁻, A[i] + A[j])
        for i ← 1 to n
            if A[i] < 0 and CONTAINS(H⁺, -A[i])
                return True
            if A[i] > 0 and CONTAINS(H⁻, -A[i])
                return True
    return False
```

**Rubric:** This question is intended to test your ability to describe an algorithm clearly and precisely. Clear and unambiguous English is instead of pseudocode is fine, but executable C/Java/Python code is not. Full credit requires correctly handling input arrays with repeated entries, including repeated zeros. Full credit for the second algorithm requires the word “expected” in the time analysis; hashing is inherently randomized!
3. Prove the following claims:

(a) For all non-negative integers \( k \), a binomial tree of order \( k \) has exactly \( 2^k \) nodes.

Solution: To simplify notation, let \( B_k \) denote the binomial tree of order \( k \), for any non-negative integer \( k \).

We prove the claim by induction. Let \( k \) be an arbitrary non-negative integer. Assume for all non-negative integers \( j < k \) that \( B_j \) has exactly \( 2^j \) nodes. There are two cases to consider.

- If \( k = 0 \), then by definition, \( B_k \) has exactly \( 1 = 2^0 \) node.
- Suppose \( k > 0 \). By definition, \( B_k \) consists of two copies of \( B_{k-1} \) joined by an edge. The induction hypothesis implies that \( B_{k-1} \) has exactly \( 2^{k-1} \) nodes. Thus, \( B_k \) has exactly \( 2 \cdot 2^{k-1} = 2^k \) nodes.

In both cases, we conclude that a binomial tree of order \( k \) has exactly \( 2^k \) nodes ■

Rubric: Full credit requires an explicit strong induction hypothesis. A proof containing a weak induction hypothesis is a Deadly Sin, and is therefore either perfect or an automatic zero. This is not the only correct proof. In particular, there are easy inductive proofs that assume either part (b) or part (c), which are worth full credit even without a solution to part (b) or part (c).

(b) For all positive integers \( k \), attaching a leaf to every node in a binomial tree of order \( k - 1 \) results in a binomial tree of order \( k \).

Solution: To simplify notation, let \( B_k \) denote the binomial tree of order \( k \), for any non-negative integer \( k \).

We prove the claim by induction. Let \( k \) be an arbitrary positive integer, and assume for all non-negative integers \( j < k \) that \( B_j \) is the result of attaching a leaf to every node in \( B_{j-1} \). There are two cases to consider.

- If \( k = 1 \), the statement is trivial: By definition, attaching a leaf (a copy of \( B_0 \)) to the only node in \( B_0 \) gives us \( B_1 \).
- Suppose \( k \geq 2 \). Let \( B^+_{k-2} \) and \( B^-_{k-2} \) be two binomial trees of order \( k - 2 \). Connecting the roots of \( B^+_{k-2} \) and \( B^-_{k-2} \) and then attaching leaves to every node yields in the same tree as attaching leaves to every node in \( B^+_{k-2} \) and \( B^-_{k-2} \) and then connecting the roots; all we’re doing is adding the edges in different orders. By the induction hypothesis, attaching a leaf to every node in \( B^+_{k-2} \) and \( B^-_{k-2} \) results in two binomial trees \( B^+_{k-1} \) and \( B^-_{k-1} \) of order \( k - 1 \). By definition, connecting the two roots of \( B^+_{k-1} \) and \( B^-_{k-1} \) gives us \( B_k \).

On the other hand, by definition, connecting the roots of \( B^+_{k-2} \) and \( B^-_{k-2} \) with an edge gives us \( B_{k-1} \). Thus, attaching leaves to every node in \( B_{k-1} \) gives us \( B_k \). In both cases, we conclude that attaching leaves to every node in \( B_{k-1} \) gives us \( B_k \). ■

Rubric: This is all about fighting the notation demons. Again, full credit requires an explicit induction hypothesis. This proof is more detailed than necessary for full credit, but any version of this argument must build up from two trees of order \( k - 2 \).

This is not the only correct proof. For example, the result also follows from the following claims for every positive \( k \): (1) Every node in \( B_k \) is either a leaf or the parent of exactly one leaf; (2) Removing every leaf from \( B_k \) yields \( B_{k-1} \).
(c) For all non-negative integers \( k \) and \( d \), a binomial tree of order \( k \) has exactly \( \binom{k}{d} \) nodes with depth \( d \).

**Solution:** For any non-negative integer \( k \), let \( B_k \) denote a binomial tree of order \( k \). For any non-negative integers \( k \) and \( d \), let \( N(k, d) \) denote the number of nodes with depth \( d \) in \( B_k \).

Let \( d \) and \( k \) be arbitrary non-negative integers. Assume that \( N(k', d') = \binom{k'}{d'} \) for all non-negative integers \( d' \) and \( k' \) such that \( k' < k \). (Notice that we are only doing induction on \( k \); the induction hypothesis considers all \( d' \))! There are three cases to consider:

- First, suppose \( d = 0 \). Every rooted tree has exactly one node with depth 0; that’s the definition of the root! Thus, \( N(k, 0) = 1 = \binom{k}{0} \).

- Now suppose \( d > 0 \) and \( k = 0 \). The binomial tree of order 0 has exactly one node, the root, which has depth 0. Thus, \( N(0, d) = 0 = \binom{0}{d} \).

- Finally, suppose \( k > 0 \) and \( d > 0 \). By definition, \( B_k \) consists of two binomial trees \( B_{k-1} \) and \( B'_{k-1} \) of order \( k - 1 \), with the root of \( B'_{k-1} \) connected as a new child of the root of \( B_{k-1} \). A node at depth \( d \) in \( B_k \) is either has depth \( d \) in \( B_{k-1} \), or it has depth \( d - 1 \) in \( B'_{k-1} \). Thus,

\[
N(k, d) = N(k - 1, d) + N(k - 1, d - 1)
\]

\[
= \binom{k-1}{d} + \binom{k-1}{d-1}
\quad\quad\quad\quad\quad\quad\quad\quad\text{[induction hypothesis]}
\]

\[
= \binom{k}{d}
\quad\quad\quad\quad\quad\quad\quad\quad\text{[Pascal's identity]}
\]

In all three cases, we conclude that \( N(k, d) = \binom{k}{d} \) for every positive integer \( d \). ■

**Rubric:** Again, full credit requires an explicit induction hypothesis and correct handling of the cases \( d = 0 \), \( k = 0 \), and \( d > k \). This is not the only correct proof. For example, you can apply induction on \( d + k \), or lexicographically on the pair \((k, d)\); independently of the precise induction hypothesis, you can use part (b) to drive the induction instead of the definition of \( B_k \). Using the factorial definition of \( \binom{k}{d} \) directly is also fine, if a bit awkward.
4. **[Extra credit]** Prove that for any arithmetic expression tree, there is an equivalent arithmetic expression tree in normal form. **[Hint: This is harder than it looks.]**

**Solution (double induction):** For any arithmetic expression trees $A$ and $B$, let $(A + B)$ denote the expression tree whose root is a $+$-node, whose left subtree is $A$, and whose right subtree is $B$. Similarly, let $(A \times B)$ denote the expression tree whose root is a $\times$-node, whose left subtree is $A$, and whose right subtree is $B$. Finally, let $f_T$ denote the function represented by the arithmetic expression tree $T$. The definitions imply immediately that $f_{(A+B)} = f_A + f_B$ and $f_{(A\times B)} = f_A \times f_B$.

**Lemma 1.** For any arithmetic expression trees $L$ and $R$ in normal form, the expression tree $(L + R)$ is in normal form.

**Proof:** Let $v$ be an arbitrary $+$-node in $(L + R)$ that is not the root. There are three cases to consider.

- If the parent of $v$ is in $L$, then it must be a $+$-node, because $L$ is in normal form.
- If the parent of $v$ is in $R$, then it must be a $+$-node, because $R$ is in normal form.
- If the parent of $v$ is the root of $(L + R)$, then it must be a $+$-node by definition.

We conclude that every $+$-node in $(L + R)$ is either the root or the child of another $+$-node. Thus, $(L + R)$ is in normal form. \(\square\)

**Lemma 2.** For any arithmetic expression trees $L$ and $R$ in normal form, there is an arithmetic expression tree in normal form that is equivalent to $(L \times R)$.

**Proof:** Let $L$ and $R$ be arbitrary arithmetic expression trees in normal form. Assume that for every proper subtree $L'$ of $L$, there is an arithmetic expression tree in normal form that is equivalent to $(L' \times R)$. There are three cases to consider:

- Suppose subtrees $L$ and $R$ have no $+$-nodes. (For example, $L$ might be a single variable node.) Then the expression tree $(L \times R)$ has no $+$-nodes, and is therefore vacuously in normal form.
- Suppose subtree $L$ contains a $+$-node. Then the root of $L$ must be a $+$-node, because otherwise $L$ would not be in normal form. Let $LL$ and $LR$ be the left and right subtrees of $L$, respectively, so $L = (LL + RR)$. Define a new expression tree $T := ((LL \times R) + (LR \times R))$. Straightforward definition-chasing implies that $T$ is equivalent to $(L \times R)$:

\[
\begin{align*}
f_T &= f_{((LL \times R) + (LR \times R))} \quad \text{[by definition of } T]\n &= f_{(LL \times R)} + f_{(LR \times R)} \quad \text{[by definition of } F_{(A+B)}]\n &= (f_{LL} \times f_R) + (f_{LR} \times f_R) \quad \text{[by definition of } F_{(A\times B)}\text{, twice]}\n &= (f_{LL} + f_{LR}) \times f_R \quad \text{[by the distributive law]}\n &= f_{LL+LR} \times f_R \quad \text{[by definition of } F_{(A+B)}]\n &= f_{((LL+LR) \times R)} \quad \text{[by definition of } F_{(A\times B)}\text{]}\n &= f_{(L \times R)}. \quad \text{[by definition of } L]\n\end{align*}
\]

Because $LL$ is a proper subtree of $L$, the induction hypothesis implies that there is a normal-form expression tree $L'$ equivalent to $(LL \times R)$. Similarly, because $LR$ is
a proper subtree of $L$, the induction hypothesis implies that there is a normal-form expression tree $R'$ equivalent to $(LR \times R)$. Define a new expression tree $T' := (L' + R')$. Because $L'$ and $R'$ are in normal form, Lemma 1 implies that $T'$ is in normal form. Straightforward definition-chasing implies that $T'$ is equivalent to $T$, and therefore equivalent to $(L \times R)$.

- Finally, suppose $R$ contains a $+$-node, but $L$ does not. Straightforward definition-chasing implies that the expression tree $(R \times L)$ is equivalent to $(L \times R)$. The previous case implies that there is an arithmetic expression tree in normal form that is equivalent to $(R \times L)$, and therefore to $(L \times R)$.

In all cases, we conclude that there is an arithmetic expression tree in normal form that is equivalent to $(L \times R)$.

\begin{proof}
Let $T$ be an arbitrary arithmetic expression tree. Assume that for any proper subtree $S$ of $T$, there is an arithmetic expression tree in normal form that is equivalent to $S$. There are three cases to consider.

- If $T$ is a single variable node, then $T$ is already in normal form.

- Suppose $T = (L + R)$ for some arithmetic expression trees $L$ and $R$. Because $L$ and $R$ are proper subtrees of $T$, the induction hypothesis implies that there is a normal-form expression tree $L'$ that is equivalent to $L$, and there is a normal-form expression tree $R'$ that is equivalent to $R$. Define a new expression tree $T' := (L' + R')$. The induction hypothesis implies that $f_{L'} = f_L$ and $f_{R'} = f_R$, and therefore

$$f_{T'} = f_{(L' + R')} = f_{L'} + f_{R'} = f_L + f_R = f_{(L + R)} = f_T.$$  

In other words, $T'$ is equivalent to $T$. Because both $L'$ and $R'$ are in normal form, Lemma 1 implies that $T' = (L' + R')$ is also in normal form.

- Finally, suppose $T = (L \times R)$ for some arithmetic expression trees $L$ and $R$. Because $L$ and $R$ are proper subtrees of $T$, the induction hypothesis implies that there is a normal-form expression tree $L'$ that is equivalent to $L$, and there is a normal-form expression tree $R'$ that is equivalent to $R$. It is easy to check that $(L' \times R')$ is equivalent to $T$, exactly as in the previous case. Lemma 2 implies that there is an expression tree $T'$ in normal form that is equivalent to $(L' \times R')$, and therefore equivalent to $T$.

In all cases, we conclude that there is an arithmetic expression tree in normal form that is equivalent to $T$. \hfill $\square$

\textbf{Theorem.} For any arithmetic expression tree $T$, there is an arithmetic expression tree in normal form that is equivalent to $T$.

\textbf{Proof:} Let $T$ be an arbitrary arithmetic expression tree. Assume that for any proper subtree $S$ of $T$, there is an arithmetic expression tree in normal form that is equivalent to $S$.

There are three cases to consider.

- If $T$ is a single variable node, then $T$ is already in normal form.

- Suppose $T = (L + R)$ for some arithmetic expression trees $L$ and $R$. Because $L$ and $R$ are proper subtrees of $T$, the induction hypothesis implies that there is a normal-form expression tree $L'$ that is equivalent to $L$, and there is a normal-form expression tree $R'$ that is equivalent to $R$. Define a new expression tree $T' := (L' + R')$. The induction hypothesis implies that $f_{L'} = f_L$ and $f_{R'} = f_R$, and therefore

$$f_{T'} = f_{(L' + R')} = f_{L'} + f_{R'} = f_L + f_R = f_{(L + R)} = f_T.$$  

In other words, $T'$ is equivalent to $T$. Because both $L'$ and $R'$ are in normal form, Lemma 1 implies that $T' = (L' + R')$ is also in normal form.

- Finally, suppose $T = (L \times R)$ for some arithmetic expression trees $L$ and $R$. Because $L$ and $R$ are proper subtrees of $T$, the induction hypothesis implies that there is a normal-form expression tree $L'$ that is equivalent to $L$, and there is a normal-form expression tree $R'$ that is equivalent to $R$. It is easy to check that $(L' \times R')$ is equivalent to $T$, exactly as in the previous case. Lemma 2 implies that there is an expression tree $T'$ in normal form that is equivalent to $(L' \times R')$, and therefore equivalent to $T$.

In all cases, we conclude that there is an arithmetic expression tree in normal form that is equivalent to $T$. \hfill $\square$
Solution (distributive law): The high-level strategy for this proof is fairly intuitive. Consider an arithmetic expression tree $T$ that is not in normal form. Let $u$ be an arbitrary +-node in $T$ whose parent $v$ is a $\times$-node. Without loss of generality, we assume that $u$ is the left child of $v$, so in the same notation as the previous proof, the subtree rooted at $v$ has the form $((LL + LR) \times R)$. Now define a new expression tree $T'$ by applying the distributive law at $v$, replacing its subtree $((LL + LR) \times R)$ with $((LL \times R) + (LR \times R))$, as shown below. An argument in the previous proof implies that $T$ and $T'$ are equivalent. We then recursively transform $T'$ into an equivalent tree in normal form.

But how do we know that this recursive algorithm actually halts?! The new tree $T'$ could have more nodes than $T$, more +-nodes than $T$, more $\times$-nodes than $T$, more +-nodes with $\times$-parents than $T$, greater depth than $T$, and so on. To make this strategy into a real proof, we define a non-negative integer potential function, with the property that applying the distributive law decreases the potential of the tree.

We define our potential function in terms of several simpler functions. The value of an expression tree $T$ is the value of its function when every variable is equal to 1; more formally, we have

$$\text{value}(T) := \begin{cases} \text{value}(A) + \text{value}(B) & \text{if } T = (A + B) \\ \text{value}(A) \cdot \text{value}(B) & \text{if } T = (A \times B) \\ 1 & \text{if } T \text{ is a single leaf} \end{cases}$$

For any node $v$ in an expression tree, let $\text{value}(v)$ denote the value of the subtree rooted at $v$. For each node $v$, let $\text{depth}^+(v)$ denote the number of ancestors of $v$ that are +-nodes, and let $\text{size}^+(v)$ denote the number of descendants of $v$ that are +-nodes. We define the potential $\Phi(v)$ of any node $v$ in an expression tree $T$ as follows:

$$\Phi(v) := 2^{\text{value}(T) - \text{depth}^+(v)}(\text{value}(v) - 1)$$

Finally, we define the potential $\Phi(T)$ of any arithmetic expression tree $T$ to be the sum of the potentials of all its $\times$-nodes. Potentials of +-nodes and leaves are ignored (and the potential of any leaf is 0 anyway).
Lemma 3. For any node \( v \) in any arithmetic expression tree, \( \text{value}(v) \geq \text{size}^+(v) + 1 \).

Proof: If \( v \) is a leaf, then \( \text{value}(v) = 1 = 1 + \text{size}^+(v) \) by definition. If \( v \) is the root of a subtree \((A + B)\), the inductive hypothesis implies that \( \text{value}(v) = \text{value}(A) + \text{value}(B) \geq \text{size}^+(A) + \text{size}^+(B) + 2 = \text{size}^+(v) + 1 \). Finally, if \( v \) is the root of a subtree \((A \times B)\), we have

\[
\begin{align*}
\text{value}(v) &= \text{value}(A) \cdot \text{value}(B) \\
&\geq \text{value}(A) + \text{value}(B) - 1 \quad (*) \\
&\geq \text{size}^+(A) + \text{size}^+(B) + 1 \quad (\dagger) \\
&= \text{size}^+(v) + 1,
\end{align*}
\]

where \((*)\) follows from the inequality \( ab - a - b + 1 = (a-1)(b-1) \geq 0 \) for all positive integers \( a \) and \( b \), and \((\dagger)\) follows from the induction hypothesis.

Corollary 1. The potential of any arithmetic expression tree is a non-negative integer.

Proof: Lemma 3 implies that \( \Phi(v) \geq 0 \) for every node \( v \). Lemma 3 also implies that \( \text{depth}^+(v) \leq \text{size}^+(T) < \text{value}(T) \), and thus \( 2^{\text{value}(T)-\text{depth}^+(v)} \) is an integer, for every node \( v \) in \( T \). We conclude that every node has integer potential. \( \square \)

Lemma 4. For any node \( v \) in any arithmetic expression tree, \( \text{value}(v) = 1 \) if and only if no descendant of \( v \) is a \(+\)-node.

Proof: If \( v \) is a leaf, then \( \text{value}(v) = 1 \) by definition.

If \( v \) is the root of a subtree \((A + B)\), Lemma 3 implies that \( \text{value}(A) \geq 1 \) and \( \text{value}(B) \geq 1 \), so \( \text{value}(v) = \text{value}(A) + \text{value}(B) \geq 2 \).

If \( v \) is the root of a subtree \((A \times B)\) that contains no \(+\)-nodes, the inductive hypothesis implies that \( \text{value}(A) = \text{value}(B) = 1 \), and therefore \( \text{value}(v) = \text{value}(A) \cdot \text{value}(B) = 1 \).

Finally, if \( v \) is the root of a subtree \((A \times B)\) that contains at least one \(+\)-node, then (without loss of generality) \( A \) must contain a \(+\)-node. Thus, the inductive hypothesis implies \( \text{value}(A) \geq 2 \) and \( \text{value}(B) \geq 1 \), and therefore \( \text{value}(v) = \text{value}(A) \times \text{value}(B) \geq 2 \). \( \square \)

Corollary 2. An arithmetic expression tree has potential 0 if and only if it is in normal form.

Proof: By definition, an arithmetic expression tree \( T \) is in normal form if and only if no \( \times\)-node in \( T \) has a \(+\)-descendant. Lemma 4 implies that a \( \times\)-node \( v \) has potential 0 if and only if \( v \) has no \(+\)-descendants. \( \square \)

Theorem. For any arithmetic expression tree \( T \), there is an arithmetic expression tree in normal form that is equivalent to \( T \).

Proof: Let \( T \) be an arbitrary arithmetic expression tree. Assume inductively that for any tree \( T' \) with \( \Phi(T') < \Phi(T) \), there is a tree in normal form equivalent to \( T' \). If \( \Phi(T) = 0 \), then \( T \) is already in normal form, by Corollary 2. So assume that \( \Phi(T) > 0 \).

Corollary 2 implies that \( T \) is not in normal form, so there must be a \(+\)-node \( u \) in \( T \) whose parent \( v \) is a \( \times\)-node. Without loss of generality, we assume that \( u \) is the left child of \( v \), so the subtree rooted at \( v \) has the form \((LL + LR) \times R\). Now define a new expression tree \( T' \) by applying the distributive law at node \( v \), replacing its subtree \((LL + LR) \times R\) with \((LL \times R) + (LR \times R)\).  

9
Straightforward definition-chasing implies that the subtrees \(((LL + LR) \times R)\) and \(((LL \times R) + (LR \times R))\) are equivalent, using the notation of the previous proof:

\[
f_{((LL \times R) + (LR \times R))} = (f_{LL} \times f_{R}) + (f_{LR} \times f_{R}) = (f_{LL} + f_{LR}) \times f_{R} = f_{((LL+LR) \times R)}.\]

It follows immediately that \(T\) and \(T'\) are equivalent.

Let \(v_L\) and \(v_R\) denote the left and right children of the root of \(((LL \times R) + (LR \times R))\), as shown in the figure above. We easily observe that \(\text{value}(v) = \text{value}(v_L) + \text{value}(v_R)\) and \(\text{depth}^+(v_L) = \text{depth}^+(v_R) = \text{depth}^+(v) + 1\), which implies that

\[
\Phi(v_L) + \Phi(v_R) = (\Phi(v) - 1)/2 < \Phi(v).
\]

Every \(\times\)-node \(w\) in the subtree \(R\) becomes two nodes \(w_L\) and \(w_R\) in \(T'\), with \(\text{value}(w_L) = \text{value}(w_R) = \text{value}(w)\) but \(\text{depth}^+(w_L) = \text{depth}^+(w_R) = \text{depth}^+(w) + 1\), and therefore \(\Phi(w_L) + \Phi(w_R) = \Phi(w)\). Every other node in \(T\) appears once in \(T'\), with the same \(\text{value}\) and \(\text{depth}^+\), and therefore the same potential.

We conclude that \(\Phi(T') < \Phi(T)\). The induction hypothesis now immediately implies that there is a normal-form tree equivalent to \(T'\), and therefore equivalent to \(T\).

Rubric: These proofs are (intentionally) more verbose than necessary for full credit. But yes, the second proof really is the shortest way I know to prove that repeatedly applying the distributive law eventually halts.

Proofs of the form “Suppose some expression tree \(T\) can be converted to normal form; now add something to \(T\) get a new tree \(T'\)...” are automatically worth zero points. That’s even worse than weak induction; it’s not induction at all!