1. Describe and analyze algorithms for the following problems. The input for each problem is an unsorted array $A[1..n]$ of n arbitrary numbers, which may be positive, negative, or zero, and which are not necessarily distinct.

   (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***:

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

```
2Sum(A[1..n]):
  seenZero ← False
  for $i \leftarrow 1$ to $n$
      if $A[i] = 0$ and seenZero
          return True
      else if $A[i] = 0$
          seenZero ← True
  $H \leftarrow$ new hash table
  for $i \leftarrow 1$ to $n$
      if $A[i] > 0$
          Insert($H, A[i]$)
  for $i \leftarrow 1$ to $n$
      if Contains($H, -A[i]$)
          return True
  return False
```

■

Rubric: Max 4 points. 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. These are not the only correct solutions. 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 must be randomized to be efficient!

(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***:

```
3Sum(A[1..n]):
  sort A
  for j ← 2 to n − 1
      i ← 1;  k ← n
      while i < j and j < k
          if A[i] + A[j] + A[k] < 0
              i ← i + 1
          else if A[i] + A[j] + A[k] > 0
              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.

```
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: Max 6 points. Full credit requires correctly handling input arrays with repeated entries, including repeated zeros. These are not the only correct solutions. As in part (a), full credit for the second algorithm requires the word "expected" in the time analysis.

2. (a) ***Practice only. No submissions will be graded.***

Describe and analyze an efficient algorithm that takes a tournament T as input and returns a Hamiltonian path in T as output.

Solution (“quicksort”, 4/5): The following recursive algorithm mirrors a classical inductive proof that every tournament contains a Hamiltonian path. The algorithm takes as input (a pointer to the adjacency matrix of) the tournament T and (an array containing indices of) a subset U of vertices, and returns as output an array containing a permutation of U that describes a simple directed path through T. The top-level call is TournamentHamPath$(T, [1, 2, \dots, n])$, where n is the number of vertices of T.

```
TournamentHamPath($T, U[1..m]$):
  if $m = 0$
      return []        ⟨⟨empty array⟩⟩
  else
      ⟨⟨Choose a pivot vertex⟩⟩
      $p \leftarrow$ any element of $S$
      ⟨⟨Partition $S \setminus p$ into predecessors and successors of $p$⟩⟩
      $U^- \leftarrow \{i \in S \mid i{\to}p$ is an edge in $T\}$
      $U^+ \leftarrow \{j \in S \mid p{\to}j$ is an edge in $T\}$
      ⟨⟨Build the output permutation⟩⟩
      $\ell \leftarrow |U^-|$
      $P[1..\ell] \leftarrow$ TournamentHamPath$(T, U^-)$
      $P[\ell+1] \leftarrow p$
      $P[\ell+2..m] \leftarrow$ TournamentHamPath$(T, U^+)$
      return $P[1..m]$
```

The definition of tournament implies that every vertex in U, except the pivot vertex p, is in exactly one of the subsets U^- and U^+.

Because T is stored as an adjacency matrix, we can check whether a given ordered pair $u{\to}v$ is an edge of T in $O(1)$ time. Thus, the total time for all *non-recursive* work in TournamentHamPath is $O(m)$. The worst-case running time of this algorithm satisfies the quicksort recurrence

$$T(m) = O(m) + \max_{\ell}\big(T(\ell) + T(m-\ell-1)\big).$$

Because $m = n$ in the top-level call, the overall algorithm runs in ***$O(n^2)$ time*** in the worst case, just like quicksort. (Moreover, this analysis is tight; in the worst case, at every level of recursion, one of the subsets U^- or U^+ is empty.) ■

If we choose the pivot vertex p *uniformly at random* from S, the resulting randomized algorithm actually runs in ***$O(n \log n)$ time with high probability***. The analysis is similar, but not identical, to the analysis for randomized quicksort, which we'll see in a few weeks. Yes, that means the running time is less than the number of edges in T with high probability.

Solution ("insertion sort", 4/5): The following iterative algorithm mirrors a classical inductive proof that every tournament contains a Hamiltonian path. Assume that the vertices of the input tournament T are (indexed by) the integers 1 through n. The algorithm takes as input (the adjacency matrix of) T, and returns as output a Hamiltonian path in T, represented as a doubly-linked list of vertices. The list P stores pointers to its first element *P.head* and its last element *P.tail*, and each vertex in the list has pointers to its successor *v.next* and its predecessor *v.prev* in P.

```
TOURNAMENTHAMPATH($T$):
  $P \leftarrow$ new list containing only vertex 1
  for $v \leftarrow 2$ to $n$
      if $v \rightarrow P.head$ is an edge in $T$
          insert $v$ at the head of $P$
      else if $P.tail \rightarrow v$ is an edge in $T$
          insert $v$ at the tail of $P$
      else
          for each vertex $u$ in $P$
              $w \leftarrow u.next$
              if $u \rightarrow v$ and $v \rightarrow w$ are edges in $T$
                  insert $v$ into $P$ between $u$ and $w$
                  break ⟨⟨exit inner for loop⟩⟩
  return $P$
```

Each linked-list insertion takes $O(1)$ time. During the vth iteration of the main for-loop, the while-loop iterates at most $O(v)$ times, so the overall algorithm runs in ***$O(n^2)$ time***. ■

If we store P in a balanced binary search tree, like an AVL or red-black tree, instead of a linked list, we can actually replace the inner for-loop with a binary-tree search that runs in $O(\log n)$ time. (This is not completely obvious; I may add more details later.) The resulting algorithm runs in ***$O(n \log n)$ time*** in the worst case. Yes, that means the running time is less than the number of edges in T! This algorithm was first sketched by Hell and Rosenfeld [1], but without the data struture details.

[1] Pavol Hell and Moshe Rosenfeld. The complexity of finding generalized paths in tournaments. *J. Algorithms* 4(4):303–309, 1983.

Solution ("mergesort", 5/5): Suppose the vertices of the input tournament T are (indexed by) the integers 1 through n. The following recursive algorithm takes as input (a pointer to the adjacency matrix of) T and two integers i and k, and returns as output a path in T through vertices i through k, represented as an array of vertices. The top-level function call is TournamentHamPath$(G, 1, n)$.

```
TournamentHamPath$(T, i, k)$:
  if $i = k$
      return $[i]$      ⟨⟨array of length 1⟩⟩
  $j \leftarrow \lfloor (i+k)/2 \rfloor$
  $L \leftarrow$ TournamentHamPath$(T, i, j)$
  $R \leftarrow$ TournamentHamPath$(T, j+1, k)$
  return MergePaths$(T, L, R)$
```

The helper function MergePaths merges the paths L and R into a single directed path that traverses the vertices of L in order and traverses the vertices of R in order. The following invariant holds at the end of each iteration of the main loop: if $i \le \ell$ and $j \le r$, then both $P[k] \to L[i]$ and $P[k] \to R[j]$ are edges in T.

```
MergePaths$(T, L[1..\ell], R[1..r])$:
  $i \leftarrow 1$;  $j \leftarrow 1$
  for $k \leftarrow 1$ to $\ell + r$
      if $j > r$
          $P[k] \leftarrow L[i]$;  $i \leftarrow i + 1$
      else if $i > \ell$
          $P[k] \leftarrow R[j]$;  $j \leftarrow j + 1$
      else if $L[i] \to R[j]$ is an edge of $T$
          $P[k] \leftarrow L[i]$;  $i \leftarrow i + 1$
      else ⟨⟨$R[j] \to L[i]$ is an edge of $T$⟩⟩
          $P[k] \leftarrow R[j]$;  $j \leftarrow j + 1$
  return $P[1..\ell + r]$
```

The helper algorithm MergePaths runs in $O(\ell + r)$ time. The main algorithm's running time obeys the standard mergesort recurrence (ignoring floors and ceilings) $T(n) = O(n) + 2T(n/2)$, so the whole algorithm runs in $\boldsymbol{O(n \log n)}$ ***time***. Yes, the running time is smaller than the number of edges in T. ■

Rubric: ***Practice only.*** An algorithm that runs in $O(n^2)$ time would be worth at most 80%. These are not the only correct algorithms; for example, Wu [1] describes a variant of heapsort that also runs in $O(n \log n)$ time. Even though these algorithms are all variants of standard sorting algorithms, it is ***not*** enough to write "sort the vertices", because the standard sorting *problem* assumes that there is a consistent underlying linear order.

[1] Jie Wu. On finding a Hamiltonian path in a tournament using semi-heap. *Inf. Proc. Lett.* 10(4):279–294, 2000.

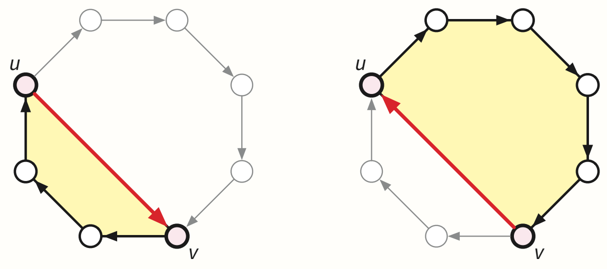
(b) Describe and analyze an efficient algorithm that takes a tournament T as input and returns as output either (1) the *only* Hamiltonian path in T or (2) a directed cycle of length 3 in T. Justify the correctness of your algorithm.

Solution (shortcuts, transitivity):

```
UniqueHamOr3cycle($T$):
    if $T$ is acyclic
        return TopologicalSort($T$)
    $C \leftarrow$ any directed cycle in $T$
    while $C$ has length greater than 3
        $u, v \leftarrow$ any two non-adjacent vertices in $C$
        if $u \to v$ is an edge in $T$
            replace subpath $u \rightsquigarrow v$ in $C$ with $u \to v$
        else
            replace subpath $v \rightsquigarrow u$ in $C$ with $v \to u$
    return $C$
```

Correctness: If T is acyclic, then it is also *transitive*: For all vertices u, v, w, if $u \to v$ and $v \to w$ are edges in T, then $u \to w$ is also an edge in T. It follows that the edges of T describe a *total* order on the vertices. Sorting the vertices according to this total order yields the only Hamiltonian path in T.

On the other hand, suppose T contains a directed cycle. Let C be the shortest such cycle. If C has length 3, we are done. Otherwise, consider any two vertices u and v of C that are not adjacent in C. The tournament contains exactly one of the edges $u \to v$ or $v \to u$. In either case, we can "shortcut" along that edge to obtain a shorter cycle, as shown below, contradicting the definition of C.



Shortcutting a directed cycle.

Running time: We can test whether T contains a cycle in $O(n^2)$ time using depth-first search. (See Chapter 6.2 of Jeff's book.) If T is acyclic, we can compute its unique Hamiltonian path in $O(n \log n)$ time using any fast sorting algorithm. Otherwise, we can obtain a directed cycle C as a byproduct of the same depth-first search. If we store C in a doubly-linked list, the entire shortcutting process takes only $O(n)$ time. The overall algorithm runs in $\boldsymbol{O(n^2)}$ ***time***. ■

Solution (quicksort, induction): The following algorithm is a modification of our first solution to part (a). The input is (a pointer to the adjacency matrix of) a tournament T. The output is either a 3-cycle whose vertices are in U, or a simple directed path in T that visits every vertex.

```
UniqueHamOr3Cycle($T$):
    if $T$ has no vertices
        return the empty path
    ⟨⟨Partition the vertices⟩⟩
    $p \leftarrow$ any vertex in $U$
    $V^- \leftarrow \{u \in V(T) \mid u \rightarrow p$ is an edge in $T\}$
    $V^+ \leftarrow \{v \in V(T) \mid p \rightarrow v$ is an edge in $T\}$
    ⟨⟨Look for 3-cycles that contain p⟩⟩
    for all $v^+ \in V^+$
        for all $v^- \in V^-$
            if $v^+ \rightarrow v^-$ is an edge in $T$
                return $p \rightarrow v^+ \rightarrow v^- \rightarrow p$
    ⟨⟨Look for 3-cycles before and after p⟩⟩
    $T^- \leftarrow$ subgraph of $T$ induced by $V^-$
    $P^- \leftarrow$ UniqueHamOr3Cycle($T^-$)
    if $P^-$ is a 3-cycle
        return $P^-$
    $T^+ \leftarrow$ subgraph of $T$ induced by $V^+$
    $P^+ \leftarrow$ UniqueHamOr3Cycle($T^+$)
    if $P^+$ is a 3-cycle
        return $P^+$
    ⟨⟨No 3-cycles found: return the unique Hamiltonian path⟩⟩
    return $P^- \rightarrow p \rightarrow P^+$
```

Running time: Let $T(n)$ denote the worst-case running time of our algorithm when the input tournament T has n vertices. Let n^- and n^+ denote the number of vertices in V^- and V^+, respectively. The worst-case running time obeys the recurrence

$$T(n) \leq \max_{n^- + n^+ = n-1} \big(T(n^-) + T(n^+) + O(n^2)\big),$$

whose solution is $\boldsymbol{T(n) = O(n^3)}$. (In the worst case, either $n^+ = 0$ or $n^- = 0$ at every level of recursion.

However, we can get a better upper bound by observing that the running time is dominated by the number of times we ask whether $u \rightarrow v$ or $v \rightarrow u$ is an edge of the input tournament. For each pair of vertices u and v, we perform this test exactly once, so the overall number of tests is exactly $\binom{n}{2}$. We conclude that our algorithm runs in $\boldsymbol{O(n^2)}$ ***time***. We can also derive this time bound by considering the more careful recurrence

$$T(n) \leq \max_{n^- + n^+ = n-1} \big(T(n^-) + T(n^+) + O(n + n^+ n^-)\big),$$

Correctness: Let T be any tournament, and suppose UNIQUEHAMOR3CYCLE(T) does not return a 3-cycle. Let $P = v_1 \to v_2 \to \cdots \to v_n$ be an ***arbtirary*** Hamiltonian path in T. We claim that UNIQUEHAMOR3CYCLE(T) must return P, which implies that P is the ***only*** Hamiltonian path in T.

First, we must have $p = u_k$ for some index k. Let P^- denote the prefix $v_1 \to v_2 \to \cdots \to v_{k-1}$ of P before the pivot vertex p, and let P^+ denote the suffix $v_{k+1} \to v_2 \to \cdots \to v_n$ of P after the pivot vertex p. (One or both of these subpaths may be empty.)

If $k = 1$, then V^- is empty, so UNIQUEHAMOR3CYCLE(T^-) must return the empty path P^-. Otherwise, we must have $v_{k-1} \in U^-$, because there are no edges from V^+ to p, and thus $v_i \in V^-$ for all $i < k$, because there are no edges *into* V^-. It follows that P^- is a Hamiltonian path in T^-. Because the recursive call UNIQUEHAMOR3CYCLE(T^-) did not return a 3-cycle, the induction hypothesis implies that P^- is the ***only*** Hamilotnian path in T^-. In particular, UNIQUEHAMOR3CYCLE(T^-) must return P^-.

A symmetric argument implies that UNIQUEHAMOR3CYCLE(T^-) must return P^+ (even if $k = n$). We conclude that UNIQUEHAMOR3CYCLE(T) must return the Hamiltonian path $P = P^- \to p \to P^+$. ■

Rubric: 10 points = 4 for algorithm + 2 for time analysis + 4 for proof of correctness. These are not the only correct solutions. These are not the only valid proofs of correctness for these algorithms.

An algorithm that runs in $O(n^3)$ time is worth at most 8 points; scale partial credit. (For example: Consider every triple of vertices; either some triple defines a cycle, or the tournament is transitive.)

3. Prove that for any arithmetic expression tree, there is an equivalent arithmetic expression tree in normal form.

Solution (double induction): For any arithmetic expression trees A and B, let $\boldsymbol{(A+B)}$ denote the expression tree whose root is a $+$-node, whose left subtree is A, and whose right subtree is B. Similarly, let $\boldsymbol{(A \times B)}$ denote the expression tree whose root is a $\times$-node, whose left subtree is A, and whose right subtree is B. Finally, let $\boldsymbol{f_T}$ 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. □

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. Without loss of generality, assume that L has more $+$-nodes than R; otherwise, we can swap the two subtrees. (Straightforward definition-chasing implies that the expression tree $(R \times L)$ is equivalent to $(L \times R)$.)

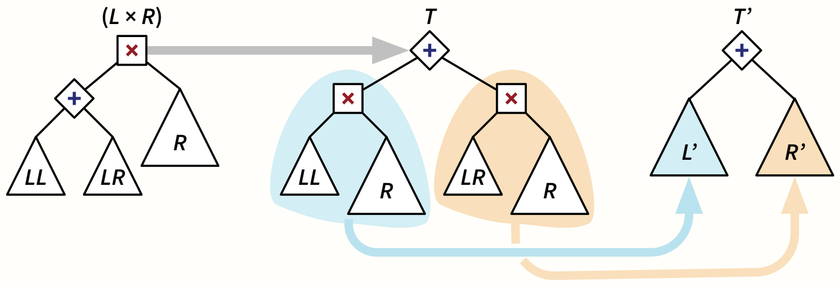
As an inductive hypothesis, assume that for every proper subtree S of L, there is an arithmetic expression tree in normal form that is equivalent to $(S \times R)$. There are two cases to consider:

- Suppose subtree L has no $+$-nodes, and therefore R has 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 $\boldsymbol{T := ((LL \times R) + (LR \times R))}$. Straightforward definition-chasing implies that T is equivalent to $(L \times R)$:

$$\begin{aligned} f_T &= f_{((LL\times R)+(LR\times R))} && \text{[by definition of } T\text{]} \\ &= f_{(LL\times R)} + f_{(LR\times R)} && \text{[by definition of } f_{(A+B)}\text{]} \\ &= (f_{LL} \times f_R) + (f_{LR} \times f_R) && \text{[by definition of } f_{(A\times B)}\text{, twice]} \end{aligned}$$

$$\begin{aligned}
&= (f_{LL} + f_{LR}) \times f_R && \text{[by the distributive law]}\\
&= f_{(LL+RR)} \times f_R && \text{[by definition of } f_{(A+B)}]\\
&= f_{((LL+LR)\times R)} && \text{[by definition of } f_{(A\times B)}]\\
&= f_{(L\times R)}. && \text{[by definition of } L]
\end{aligned}$$

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 $\boldsymbol{T' := (L' + R')}$. 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 $(L \times R)$.



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

Now we are ready to prove the main theorem. 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 are normal-form expression trees L' and R' that are equivalent to L and R, respectively. Define a new expression tree $\boldsymbol{T' := (L' + R')}$. The definition of equivalnce implies that $f_{L'} = f_L$ and $f_{R'} = f_R$, and therefore

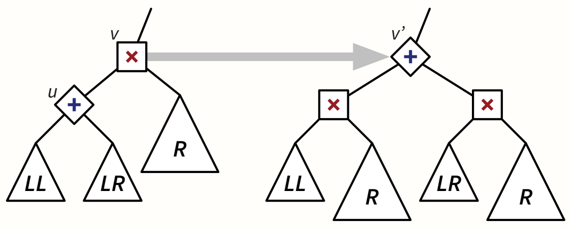
$$f_{T'} = f_{(L'+R')} = f_{L'} + f_{R'} = f_L + f_R = f_{(L+R)} = f_T.$$

  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 are normal-form expression trees L' and R' that are equivalent to L and R, respectively. Straightforward definition-chasing implies 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. ■

Solution (potential function): 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 can assume that u is the left child of v, so in the same notation as the previous solution, 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. Straightforward definition-chasing implies that T and T' are equivalent; see Lemma 2 in the previous proof. We then recursively transform T' into an equivalent tree in normal form.



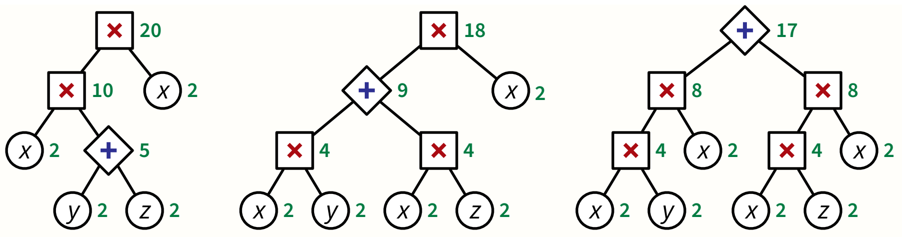
Applying the distributive law.

But the new tree T' isn't always "smaller" than T. Specifically, 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. **How do we know that our simple algorithm—repeatedly apply the distributive law—actually halts?!**

To prove that our iterative improvement strategy is a real algorithm, we define a positive integer *potential* for any expression tree, with the property that applying the distributive law *always* decreases potential. We define our potential function recursively as follows:

$$\Phi(T) = \begin{cases} 2 & \text{if } T \text{ is a single leaf} \\ \Phi(L) + \Phi(R) + 1 & \text{if } T = (L + R) \\ \Phi(L) \cdot \Phi(R) & \text{if } T = (L \times R) \end{cases}$$

Our main claim is that if we apply the distributive law anywhere in any expression tree T to obtain a new tree T', then $\Phi(T') < \Phi(T)$. Because potential is always a positive integer, we can only decrease it a finite number of times.



Three equivalent expression trees with different potentials.
The number next to each node is the potential of the subtree rooted at that node.

Lemma 3. *For every arithmetic expression tree T, we have $\Phi(T) \geq 2$.*

Proof: Let T be any arithmetic expression tree. Assume for every proper subtree S of T that $\Phi(S) \geq 2$. There are three cases to consider.

If T is a leaf, then $\phi(v) = 2$ by definition.

If $T = (L + R)$ for some expression trees L and R, the definition of Φ and the induction hypothesis imply that $\Phi(T) = \Phi(L) + \Phi(R) + 1 \geq 2 + 2 + 1 > 2$.

Finally, if $T = (L \times R)$ for some expression trees A and B, the definition of Φ and the induction hypothesis imply that $\Phi(T) = \Phi(L) \cdot \Phi(R) \geq 2 \cdot 2 > 2$. □

Lemma 4. *For any expression trees LL, LR, and R, we have $\Phi((LL + LR) \times R) > \Phi((LL \times R) + (RR \times R))$.*

Proof: This is mostly definition-chasing:

$$\begin{aligned}
&\Phi((LL + LR) \times R) \\
&\qquad = \Phi(LL) \cdot \Phi(R) \ + \ \Phi(LR) \cdot \Phi(R) \ + \ \Phi(R) && \text{by definition of } \Phi \\
&\qquad > \Phi(LL) \cdot \Phi(R) \ + \ \Phi(LR) \cdot \Phi(R) \ + \ 1 && \text{by Lemma 3} \\
&\qquad = \Phi((LL \times R) + (RR \times R)) && \text{by definition of } \Phi
\end{aligned}$$
□

Lemma 5. *Let T be any expression tree, let S be any subtree of T, let S' be any expression tree, and let T' be the expression tree obtained from T by replacing S with S'. If $\Phi(S') < \Phi(S)$, then $\Phi(T') < \Phi(T)$.*

Proof: Straightforward definition-chasing. □

Now we are ready to prove the main result. Let T be an arbitrary arithmetic expression tree. As an inductive hypothesis, assume that for any tree T' with $\Phi(T') < \Phi(T)$, there is a normal-form expression tree equivalent to T'.

If T is already in normal form, the theorem is trivial. Otherwise, without loss of generality, T contains a subtree S of the form $((LL + LR) \times R)$. Define a new expression tree T' by applying the distributive law, replacing S with $S' = ((LL \times R) + (LR \times R))$. We've already shown that the old tree T and the new tree T' are equivalent.

Lemma 4 implies $\phi(S') < \Phi(S)$, so Lemma 5 implies $\Phi(T') < \Phi(T)$. The induction hypothesis now immediately implies that there is a normal-form tree equivalent to T', and therefore equivalent to T. ■

Rubric: Max 10 points. These proofs are (intentionally) more verbose than necessary for full credit. These are not the only correct proofs. In particular, the potential function used in the second proof is not the only useful potential function (but it's the simplest one I know).

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