factorCheck

Checks provided factoring to see if it matches author's factoring.

JavaScript

```

var debugOn=true;

/*
HOW THIS SHOULD WORK:
    Initially check to make sure the submitted answer (and proposed answer) are at least in some kind of theo

    Next duplicate the raw trees so we can mess with them without worrying about changing the original.

    Fold up any exponents on both trees so that our root node is of the form: ['*',factor1,factor2,factor3,..
        This includes killing off any leading negative signs (again we'll compare for equality using the orig

    Now call a recursive function to deep-dive into each factor to find what degree that factor actually is.
        Will also identify if the factor isn't even a factor, in which case we will return a negative value t

    Once we have the degree for each factor, now we can compare the instructor degree list and student degree
*/


// Subfunction to identify if something is a number:

function isNum(numb) {
    if ((typeof numb === 'number')||(numb=='e')||(numb=='pi'))
    {return true} else {return false}
}

// Subfunction to identify if something is a non-negative integer:

function isPosInt(numb) {
    if ((isNum(numb))&&((numb>=0)&&(numb%1==0)))
    {return true} else {return false}
}


// This does a recursion through a factor to eventually find it's degree - assuming it's a polynomial.

function degreeHunt(tree,position,curDeg) {
    //(Re)set curDeg just in case:
    var curDeg=0;

    // First, let's figure out what to do about negative signs, since they can be annoying.
    //  I think there's three possibilities, it's a negative array, a negative x, or a negative number.

    if (tree[position][0]=='-') {
        debugText('Processing a minus sign.');
        if (tree[position][1]=='x') {
            // We found a ``-x'' term within our factor, so that's degree 1 I guess!

            debugText('Found a -x term!');
            curDeg = Math.max(curDeg,1);
```

Learning outcomes:

```

        } else if (Array.isArray(tree[position][1])) {
            // Else if there's a negative outside of an array, just bypass the negative and keep digging for

            debugText('Found a negative Array term!');
            let tempVal = degreeHunt(tree[position],1)
            if (tempVal<0) { return (-1)} else {
                curDeg = Math.max(curDeg,tempVal);
            }


        }// Note that, if it's a negative number, I don't care about it, so no need for an ``else''.
    } else if (tree[position][0]=='apply') {
        debugText('Processing an apply symbol.');
        // any 'apply' is inevitably a function that isn't a polynomial.
        //  Although, only if it actually has an 'x' in there - so I need to fix/update that at some point.

        // To find out if the apply is actually just a number or not, we need to recurs through every sub-nod
        //  If the degree ends up positive, then we have a variable inside the apply function it's not a poly
        //  If the degree ends up zero, then it's ultimately some bizarre number formation and we're fine.

        var tempDeg = 0;

        for (var j = 1; j < tree[position].length; ++j) {
            // Walk the array to find any powers of x.
            if (tree[position][j]=='x') {
                // If the entry is just x, then we have pos deg and we are done.
                debugText('Found an x inside an apply function that suggests the factor is not actually a pol
                return (-1);
            } else if (Array.isArray(tree[position][j])) {
                // If the term is an array, then we need to recurs to find the degree.
                let tempVal=degreeHunt(tree[position],j,0);
                if (tempVal!==0) {
                    debugText('Found an issue inside an apply function that suggests the factor is not actual
                    return (-1)
                }
            }// Any other option is degree 0, so no need for an else.
        }// End of for loop and end up 'apply' function
    } else if (tree[position][0]=='^') {
        // Now we process the exponent sign case, but be careful cause students do crazy shit,
        //  So it might be a x^N situation, but it might be some other shenanigans.

        debugText('Processing an exponential sign.');

        if ((tree[position][1]=='x')&&(isPosInt(tree[position][2]))) {
            // We have something like x^N

            debugText('We have x^N');

            curDeg = Math.max(curDeg,tree[position][2]);

        } else if (tree[position][1]=='x') {
            // If the base is x but it's not being raised to an integer power, then it's not a proper monomia
            //  Note that we are assuming here that students won't put shit like ``1+1'' as the power, if so,

            debugText('We have x^(g(x)) but g(x) is not a positive integer.');

            return -1;

        } else if ((isNum(tree[position][1]))&&(isNum(tree[position][2]))) {
            // This means we have a^b which is still just a number, so it’s fine... but doesn’t give a degree

            debugText(’We have a^b’);

        } else if ((Array.isArray(tree[position][1]))&&(isPosInt(tree[position][2]))) {
            // We have something of the form (f(x))^N which might be a part of a factor.
            //  We recurse on the array, and multiply the result by the N.

            debugText(’We have f(x)^N’);
            let tempVal = tree[position][2]*degreeHunt(tree[position],1,curDeg);
            if (tempVal<0) { return (-1)} else {
                curDeg = Math.max(curDeg,tempVal);
            }


        } else {
            // All other situations are inevitably not polynomials.

            return (-1);
        }
    } else if (tree[position][0]==’/’) {
        // This is problematic, because the only way we can allow a division is if the denominator is just a
        //  But if the denominator has an ’x’ anywhere in it, then we have a problem...

        debugText(’Processing a division sign.’);

        if (tree[position][2]==’x’) {
            // We’re dividing by x, which is bad.

            debugText(’Dividing by x, naughty naughty!’);

            return (-1);
        } else if (Array.isArray(tree[position][2])) {
            //  Let’s try doing a recurs, and if we get a result that isn’t 0, then that means there’s an ’x’
            if (degreeHunt(tree[position],2,0)!==0) {
                // If it’s not 0, then we found an ’x’ or something that invalidates the polynomial.
                return (-1)
            } // otherwise we have some kind of ’dividing by a number’ situation, so it’s fine, and doesn’t i

        }

        // Now that we’ve dealt with the possibilities that cause a non-polynomial function, we can proceed a

        debugText(’The denominator is just a number, so we need to check the numerator.’);

        if (tree[position][1]==’x’) {
            // If we’ve made it past the first two hurtles, then the bottom is just a number of some form.
            //  So if the top is ’x’, then we have something like ‘‘x/a’’ which is still a degree 1 factor.
            var curDeg = Math.max(curDeg,1);
            debugText(’We found a fraction with just x in the top.’);
        } else if (Array.isArray(tree[position][1])) {
            // If the top of the fraction is an array - but we’ve already reduced to case where denominator i
            //  So we need to figure out if there is a degree in the top to count.

            debugText(’We found a fraction with an array for the numerator.’)
            var tempVal = degreeHunt(tree[position],1,0)
            debugText(’I think the numerator degree from the array is: ’+tempVal);
            if (tempVal<0) { return (-1)} else {
                 var curDeg = Math.max(curDeg,tempVal);
            }

        }// Note that if none of the above, then it's just a/b, which is fine and doesn't impact degree.

    } else if (tree[position][0]=='*') {
        // If we are multiplying, we could be multiplying a bunch of terms - maybe a bunch of x terms.

        debugText('Processing a product sign.');

        var tempDeg = 0;
        for (var j = 0; j < tree[position].length; ++j) {
            // Add the degree of each thing being multiplied, even though most are probably zero.
            if (tree[position][j]=='x') {
                // If the entry is just x, then we have deg 1.
                debugText('Found a solo x term inside the product sign, degree is at least 1.')
                tempDeg = Math.max(tempDeg,1);
            } else if (Array.isArray(tree[position][j])) {
                // If the term is an array, then we need to recurs to find the degree.
                // let tempVal=degreeHunt(tree[position],j,0);
                if (degreeHunt(tree[position],j,0)<0) {
                    return (-1)
                } else if (degreeHunt(tree[position],j,0)==0){
                    debugText('No degree term found inside product sign, so we stay at degree: '+tempDeg);
                } else {
                    debugText('found a higher degree term inside the product sign, degree is at least: '+temp
                    tempDeg = Math.max(tempDeg,degreeHunt(tree[position],j,0));
                }
            }// Any other option is degree 0, so no need for an else.
        }
        // Once the for loop finishes, we convert the temp deg to the current deg.
        var curDeg = Math.max(curDeg,tempDeg);
        debugText('Inside the product sign we found the degree is: '+tempDeg);
        debugText('So we set the curDeg to: '+curDeg);
    } else {
        // If it's none of the above, then we should just recurs on any arrays we find.

        debugText('Processing an unknown sign? Specifically: '+tree[position][0]);

        for (var j = 0; j < tree[position].length; ++j) {
            // Walk the array to find any powers of x.
            if (tree[position][j]=='x') {
                // If the entry is just x, then we have deg 1.
                curDeg = Math.max(curDeg,1);
            } else if (Array.isArray(tree[position][j])) {
                // If the term is an array, then we need to recurs to find the degree.
                let tempVal=degreeHunt(tree[position],j,0);
                if (tempVal<0) { return (-1)} else {
                    curDeg = Math.max(curDeg,tempVal);
                }
            }// Any other option is degree 0, so no need for an else.
        }

    }
    debugText('Made it to the end of the degreeHunt function, which means we need to return a curDeg variable
    return curDeg
}

// Subfunction just to make sure that the submitted function is in a legitimately factored form.
```

```

function JNFisFactored(factorTree) {

    // First we check to see if we have a negative factored out, which messes everything up in the tree.
    if ((factorTree[0]=='-')||(factorTree[0]=='*')||((factorTree[0]=='/')&&(isNum(JNFoperation[1])))
    ) {return true} else {return false}
}

// Subfunction to make debug easier.

function debugText(text) {
    if (debugOn) {
    console.log('DEBUG INFO::' + text)
    }
}



function factorCheck(f,g) {
    // This validator is designed to check that a student is submitting a factored polynomial. It works by:
    //  Checking that the degree of each factor matches between student submitted and instructor submitted an
    //  Checking that the submitted answer and the expected answer are the same via real Xronos evaluation,
    //  Checking that the outer most (last to be computed when following order of operations) operation is mu
    //  It ignores degree 0 terms for degree check, and now can ignore factored out negative signs.


    console.log(f.tree);
    console.log(g.tree);

    if (JNFisFactored(g.tree)==false) {
        console.log('Answer rejected, instructor answer not in a factored form. Bad instructor, no donut.');
        return false
    }

    // First we check to make sure it is in *a* factored form:
    if (JNFisFactored(f.tree)==true) {
        console.log('The student answer is at least in *a* factored form.');


    } else {
        console.log('Answer rejected, student answer not in a factored form.');
        return false
    }

    // Let's duplicate the trees to manipulate, so we keep the original correctly.
    var studentAns=f.tree
    var instructorAns=g.tree

    // Also, if there is a factored out negative, let's just kill that, since we aren't doing a funtion compa
    while (studentAns[0]=='-'){studentAns = studentAns[1];};
    while (instructorAns[0]=='-'){instructorAns = instructorAns[1];};

    // Now we want to fold up any root-level exponents into duplicate children of the master tree,
    //  This lets us assume the top-level node has 1 child per factor.
    debugText('folding up external exponents of studentTree  so factors do not have exponents')
    for (var i = 0; i < studentAns.length; ++i) {
        if ((studentAns[i][0] == '^')&&(isPosInt(studentAns[i][2]))) {
            studentAns=studentAns.concat(Array(studentAns[i][2]).fill(studentAns[i][1]));
            studentAns.splice(i,1);// This should theoretically remove the original term now that we've dupli
            i=i-1;// since we shortened our array by 1, we should move the iteration value down 1 too.
            debugText('Ok, I folded up a term, so hopefully our student vector still makes sense. It is now:
        } else if (studentAns[i][0] == '^') {
            //if we have a power, but not a positive integer power, then we have a non-polynomial factor, so
            console.log('I think I found a non-polynomial term, specifically some root term has a non natural
            return false;
        }
    }
    debugText('After all preprocessing my studentAns vector is:');
    console.log(studentAns);

    // Now re repeat with instructor tree:
    debugText('folding up external exponents of instructorTree so factors do not have exponents')
    for (var i = 0; i < instructorAns.length; ++i) {
        if ((instructorAns[i][0] == '^')&&(isPosInt(instructorAns[i][2]))) {
            instructorAns=instructorAns.concat(Array(instructorAns[i][2]).fill(instructorAns[i][1]));
            instructorAns.splice(i,1);// This should theoretically remove the original term now that we've du
            i=i-1;// since we shortened our array by 1, we should move the iteration value down 1 too.
            debugText('Ok, I folded up a term, so hopefully our student vector still makes sense. It is now:
        } else if (instructorAns[i][0] == '^') {
            //if we have a power, but not a positive integer power, then we have a non-polynomial factor, so
            return false;
            console.log('Found a non-polynomial term in the instructor answer... huh? Check the code!');
        }
    }
    debugText('After all preprocessing my instructorAns vector is: '+instructorAns);

    /*
        ::NOW LETS PROCESS THE STUDENT ANSWER::
    */

    var studentDegList=[0]
    for (var i = 0; i < studentAns.length; ++i) {
        if (studentAns[i] == 'x') {
            // If the factor is simply 'x', then it's a degree 1 factor... yay?
            studentDegList.push(1);
            debugText('Found another factors degree, so now studentDegList is: '+studentDegList);
        } else if (Array.isArray(studentAns[i])) {
            // Otherwise, if it is an array, we have something to go hunting in.
            studentDegList.push(degreeHunt(studentAns,i,0));
            debugText('Found another factors degree, so now studentDegList is: '+studentDegList);
        }// Note that the only other possibility is it being a number, which we don't care about.
    }
    studentDegList = studentDegList.filter(x => x!==0);// Remove all zeros from the array to avoid stupid pad
    studentDegList.sort();// Sort the result so that we can later compare it to the instructor version.
    debugText('The final List of Factor Degrees given by the student is: ' + studentDegList);

    if (studentDegList.some(elem => elem<0)) {
        console.log('I think one of the student factors is NOT a polynomial. So I am rejecting the answer.');
        return false
        }


    /*
        ::NOW LETS PROCESS THE INSTRUCTOR ANSWER::
    */

    var instructorDegList=[0]
    for (var i = 0; i < instructorAns.length; ++i) {
        if (instructorAns[i] == 'x') {
            // If the factor is simply 'x', then it's a degree 1 factor... yay?
            instructorDegList.push(1);
            debugText('Found another factors degree, so now instructorDegList is: '+instructorDegList);
        } else if (Array.isArray(instructorAns[i])) {
            // Otherwise, if it is an array, we have something to go hunting in.
            instructorDegList.push(degreeHunt(instructorAns,i,0));
            debugText('Found another factors degree, so now instructorDegList is: '+instructorDegList);
        }// Note that the only other possibility is it being a number, which we don't care about.
    }
    instructorDegList = instructorDegList.filter(x => x!==0);// Remove all zeros from the array to avoid stup
    instructorDegList.sort();// Sort the result so that we can later compare it to the instructor version.
    debugText('The final List of Factor Degrees given by the instructor is: ' + instructorDegList);

    if (instructorDegList.some(elem => elem<0)) {
        console.log('I think one of the instructor factors is NOT a polynomial. So I am rejecting the answer.
        return false
        }

   /*
        ::NOW WE COMPARE::
   */

    if (studentDegList.length!=instructorDegList.length){
        console.log('Ans Rejected: Wrong number of factors.');
        return false;
        }

    for (var i = 0; i < studentDegList.length; ++i) {
        if (studentDegList[i] !== instructorDegList[i]) {
        console.log('Ans Rejected: At least one factor is the wrong degree.');
        return false;
        }
    }

    if (f.equals(g)){
        } else {
        console.log('Ans Rejected: Factors do not expand to original Polynomial.');
        }

    return (f.equals(g));
}
```

Problem 1 *This validator checks to see if the provided "factored form" from the student is actually factored in a similar way to the author-provided "factored form".*

For example, if you have the polynomial $x^3 - 4x^2 - 4x + 16$, you might want the student to just do the factor by grouping step and want them to enter in $(x^2 - 4)(x - 4)$. Try trying in the full polynomial versus a fully factored version, versus the desired version: $\boxed{(x^2 - 4)(x - 4)}$

Problem 2 *Now, let's say you want them to fully factor, not just factor by grouping. Try the factor by grouping version versus unfactored vs fully factored here:* $\boxed{(x - 2)(x + 2)(x - 4)}$

Problem 3 *Adding another problem here that deliberately has some repeated factors to test out if they work. The answer should be* $(x-1)^3(x+1)(x^2-1)$ *(Also testing the 'not fully factored version is the target to accept').*

$\boxed{(x-1)^3(x+1)(x^2-1)}$

Problem 4 *Finally, made a sage generated version so that we can make sure nothing about sage syntax messes things up.*
Actual answer should be $-3\,(5\,x-3)(4\,x+3)(3\,x-1)(2\,x+3)(x+3)(x+2)x^4$. *[Also stress testing to make sure larger number of factors isn't an issue.]*

$\boxed{-3\,(5\,x-3)(4\,x+3)(3\,x-1)(2\,x+3)(x+3)(x+2)x^4}$

Potential Problems and Pitfalls

The current generation of Xronos *really* isn't designed to have this level of custom validation check - so the actual validator code is remarkably hacky and intensely exploits how the data was saved in the backend of the renderer *at the time I wrote the validator.* The current generation of this validator is fairly robust, but future patches to underlying systems may break it. Currently, as long as numeric exponents are actually simplified (e.g. students write an exponent as 2 instead of 1+1) things seem to work pretty much as expected (including correctly handling negative signs, simplified exponents, and fractions). Some irrational and weird numbers might cause issues, but that is more to do with needing to figure out how numbers in weird formats might be submitted or encoded and I don't have enough data for that yet.