Algorithm 1 JDR algorithm

```
function DIFF(old, new)
    patches ← NULL
    commonSubtree ← NULL
    path ← root
    genCommonSubtree(old, new, commonSubtree, path, patches)
    genPatch(old, new, commonSubtree, path, patches)
end function
function GENCOMMONSUBTREE(old, new, commonSubtree, path, patches)
    if deepEqual(old, new) then
        commonSubtree.push(path, JSON.stringify(new))
        return
    end if
    if typeOf(old) ≠ typeOf (new) then
        return
    end if
    if Array.isArray(old) & Array.isArray(new) then
        return
    end if
    if Object.isObject(old) & Object.isObject(new) then
        commonObjectNode(old, new, commonSubtree, path)
        return
    end if
     return genLeafPatch(old, new, commonSubtree, path, patches);
end function
function COMMONOBJECTNODE(old, new, commonSubtree, path)
    oldSubTrees ← Object.subtrees(old);
    newSubTrees← Object.subtrees(new);
    for each oldSubTree in oldSubTrees do
        if new.hasNode(oldSubTree) then
            genCommonSubtree(old.oldSubTree, new.oldSubTree, commonSubtree, oldSub-
Tree.path)
        end if
    end for
end function
function GENLEAFPATCH(old, new, commonSubtree, path, patches)
    if new ≠ old then
        patches.push("replace", path: path, value: new)
    end if
end function
function GENPATCH(old, new, commonSubtree, path, patches)
    if Array.isArray(old) & Array.isArray(new) then
        arrayOperationalTransformation(old, new, commonSubtree, path, patches)
        return
    end if
    if Object.isObject(old) & Object.isObject(new) then
        genObjectPatch(old, new, commonSubtree, path, patches)
        return
    end if
     return genLeafPatch(old, new, commonSubtree, path, patches)
end function
function GENOBJECTPATCH(old, new, commonSubtree, path, patches)
    oldSubTrees ← Object.subtrees(old);
    newSubTrees ← Object.subtrees(new);
    for each oldSubTree in oldSubTrees do
        if new.hasNode(oldSubTree) then
            genPatch(old.oldSubTree, new.oldSubTree, commonSubtree, oldSubTree.path)
        else
            patches.push("remove", path: oldSubTree.path, value: old.oldSubTree)
        end if
    end for
    for each newSubTree in newSubTrees do
        newVal ← new.newSubTree.value
        if !old.hasNode(newSubTree) then
            if commonSubtree.equalValNode(newVal) then
                patches.push("copy", path: newSubTree.path, from: equalValNodePath)
            else if pathes.equalValRemovedNode(newVal) then
                patches.push("move", path: newSubTree.path, from: equalValRemovedNodePath)
            else
                patches.push("add", path: newSubTree.path, value: newVal)
            end if
        end if
    end for
end function
```

Algorithm 2 Array Operational Transformation

```
a ← [a_0, a_1, .., a_p, .., a_m]
b ← [b_0, b_1, .., b_q, .., b_n]
a ← Hash(a)                                   ▷ Hash each object in the array.
b ← Hash(b)                                   ▷ b_q = {H_q, V_q, I_q}
a ← Sort(a)                                   ▷ Sort the array according to the H of each object.
b ← Sort(b)                                   ▷ b_j = {H_q, V_q, I_q}
i = 0; j = 0; tmppatch ← [ ];
while i < m do                                ▷ Compare two array and get temperate patch.
    while j < n do
        if a[i].H > b[j].H then
            tmppatch ← tmppatch + {''add'', H_q, V_q, I_q}
            j ← j + 1
        else if a[i].H < b[j].H then
            tmppatch ← tmppatch + {''remove'', H_p, V_p, I_p}
            i ← i + 1
        else
            tmppatch ← tmppatch + {''move'', H_p, V_p, I_p → I_q}
            i ← i + 1
            j ← j + 1
        end if
    end while
    if i < m then
        tmppatch ← tmppatch + {''remove'', H_p, V_p, I_p}
        i ← i + 1
    end if
end while
tmppatch ← Sort(tmppatch)                     ▷ Sort the array according to origin index.
patch ← [ ]; arrCommon ← [ ];
for each op in tmppatch do
    op.I_p ← transformIndex(patch, op)
    switch op.type do
        case add
            if x ← findCopy(op, patch, arrCommon) then
                patch ← patch + {''copy'', x.I_q → op.I_q}
            else if findReplace(op, patch) = True then
                patch.pop()
                patch ← patch + {''replace'', op.V_q, op.I_q}
            else
                patch ← patch + {''add'', op.V_q, op.I_q}
            end if
        case remove
            patch ← patch + {''remove'', op.I_p}
        case move
            if op.I_p = op.I_q then
                arrCommon ← arrCommon + op
            end if
            patch ← patch + {''move'', op.I_p → op.I_q}
end for
function transformIndex(patch, op)
    finalIndex ← 0
    if op.type is ''add'', ''replace'' or ''copy'' then
        finalIndex ← op.I_q
    else
        for each previousOp in patch do
            switch previousOp.type do
                case add and copy
                    if finalIndex >= previousOp.I_q then
                        finalIndex ← finalIndex + 1
                    end if
                case replace
                    finalIndex ← finalIndex
                case remove
                    if finalIndex > previousOp.I_p then
                        finalIndex ← finalIndex − 1
                    end if
                case move
                    if previousOp.I_p ≠ previousOp.I_q then
                        if finalIndex ∈ Interval(previousOp.I_p, previousOp.I_q) then
                            if previousOp.I_p > previousOp.I_q then
                                finalIndex ← finalIndex + 1
                            else
                                finalIndex ← finalIndex − 1
                            end if
                        end if
                    end if
        end for
    end if
    return finalIndex
end function
```