8021. Minimum Operations to Form Subsequence With Target Sum

My Submissions (/contest/weekly-contest-360/problems/minimum-operations-to-form-subsequence-with-target-sum/submissions/)

Back to Contest (/contest/weekly-contest-360/)

You are given a **0-indexed** array `nums` consisting of **non-negative** powers of `2`, and an integer `target`.

In one operation, you must apply the following changes to the array:

- Choose any element of the array `nums[i]` such that `nums[i] > 1`.
- Remove `nums[i]` from the array.
- Add **two** occurrences of `nums[i] / 2` to the **end** of `nums`.

Return the ***minimum number of operations*** *you need to perform so that* `nums` *contains a* ***subsequence*** *whose elements sum to* `target`. If it is impossible to obtain such a subsequence, return `-1`.

A **subsequence** is an array that can be derived from another array by deleting some or no elements without changing the order of the remaining elements.

User Accepted:	126
User Tried:	475
Total Accepted:	127
Total Submissions:	766
Difficulty:	Medium

Example 1:

```
Input: nums = [1,2,8], target = 7
Output: 1
Explanation: In the first operation, we choose element nums[2]. The array becomes equal to nums = [1,2,4,4].
At this stage, nums contains the subsequence [1,2,4] which sums up to 7.
It can be shown that there is no shorter sequence of operations that results in a subsequnce that sums up to 7.
```

Example 2:

```
Input: nums = [1,32,1,2], target = 12
Output: 2
Explanation: In the first operation, we choose element nums[1]. The array becomes equal to nums = [1,1,2,16,16].
In the second operation, we choose element nums[3]. The array becomes equal to nums = [1,1,2,16,8,8]
At this stage, nums contains the subsequence [1,1,2,8] which sums up to 12.
It can be shown that there is no shorter sequence of operations that results in a subsequence that sums up to 12.
```

Example 3:

```
Input: nums = [1,32,1], target = 35
Output: -1
Explanation: It can be shown that no sequence of operations results in a subsequence that sums up to 35.
```

Constraints:

- `1 <= nums.length <= 1000`
- `1 <= nums[i] <= 2^30`
- `nums` consists only of non-negative powers of two.
- `1 <= target < 2^31`

JavaScript

```
class SplayNode {
    constructor(value) {
        this.parent = null;
        this.left = null;
        this.right = null;
        this.val = value;
        this.sum = value;
        this.sz = 1;
    }
    update() {
        this.sz = (this.left != null ? this.left.sz : 0) + (this.right != null ? this.right.sz : 0) + 1;
        this.sum = (this.left != null ? this.left.sum : 0) + (this.right != null ? this.right.sum : 0) + this.val;
```

```
    }
    isLeft() {
        return this.parent != null && this.parent.left == this;
    }
    isRight() {
        return this.parent != null && this.parent.right == this;
    }
    isRoot(guard = null) {
        return this.parent == guard;
    }
}

// MultiSet
class SplayTree {
    constructor() {
        this.root = null;
        this.cmp = (x, y) => x >= y ? 0 : 1;
    }
    zig(x) { // right rotation
        let y = x.parent;
        if (x.right != null) x.right.parent = y;
        y.left = x.right;
        x.right = y;
        if (y.isLeft()) {
            y.parent.left = x;
        } else if (y.isRight()) {
            y.parent.right = x;
        }
        x.parent = y.parent;
        y.parent = x;
        y.update();
        x.update();
    }
    zag(x) { // left rotation
        let y = x.parent;
        if (x.left != null) x.left.parent = y;
        y.right = x.left;
        x.left = y;
        if (y.isLeft()) {
            y.parent.left = x;
        } else if (y.isRight()) {
            y.parent.right = x;
        }
        x.parent = y.parent;
        y.parent = x;
        y.update();
        x.update();
    }
    zigzig(x) { // RR
        this.zig(x.parent);
        this.zig(x);
    }
    zigzag(x) { // RL
        this.zig(x);
        this.zag(x);
    }
    zagzag(x) { // LL
        this.zag(x.parent);
        this.zag(x);
    }
    zagzig(x) { // LR
        this.zag(x);
        this.zig(x);
    }
    splay(node, guard = null) { // splay node under guard, default splay to root
        while (!node.isRoot(guard)) {
            if (node.parent.isRoot(guard)) {
                if (node.isLeft()) {
                    this.zig(node);
                } else {
                    this.zag(node);
                }
            } else {
                if (node.parent.isLeft()) {
                    if (node.isLeft()) {
```

```
                        this.zigzig(node);
                    } else {
                        this.zagzig(node);
                    }
                } else {
                    if (node.isRight()) {
                        this.zagzag(node);
                    } else {
                        this.zigzag(node);
                    }
                }
            }
        }
        if (guard == null) this.root = node;
    }
    LastNode(x) {
        this.splay(x);
        let node = x.left;
        if (node == null) return null;
        while (node.right != null) node = node.right;
        this.splay(node);
        return node;
    }
    NextNode(x) {
        this.splay(x);
        let node = x.right;
        if (node == null) return null;
        while (node.left != null) node = node.left;
        this.splay(node);
        return node;
    }
    find(value) {
        return this.findFirstOf(value);
    }
    findFirstOf(value) {
        let node = this.root, res = null, last_visited = null;
        while (node != null) {
            last_visited = node;
            if (this.cmp(value, node.val)) {
                node = node.left;
            } else if (this.cmp(node.val, value)) {
                node = node.right;
            } else {
                res = node;
                node = node.left;
            }
        }
        if (last_visited != null) this.splay(last_visited);
        return res;
    }
    findLastOf(value) {
        let node = this.root, res = null, last_visited = null;
        while (node != null) {
            last_visited = node;
            if (this.cmp(value, node.val)) {
                node = node.left;
            } else if (this.cmp(node.val, value)) {
                node = node.right;
            } else {
                res = node;
                node = node.right;
            }
        }
        if (last_visited != null) this.splay(last_visited);
        return res;
    }
    findRankOf(node) {
        this.splay(node);
        return node.left == null ? 0 : node.left.sz;
    }
    findSuccessorOf(value) {
        let node = this.root, res = null, last_visited = null;
        while (node != null) {
            last_visited = node;
            if (this.cmp(value, node.val)) {
```

```
                res = node;
                node = node.left;
            } else {
                node = node.right;
            }
        }
        if (last_visited != null) this.splay(last_visited);
        return res;
    }
    findPrecursorOf(value) {
        let node = this.root, res = null, last_visited = null;
        while (node != null) {
            last_visited = node;
            if (this.cmp(node.val, value)) {
                res = node;
                node = node.right;
            } else {
                node = node.left;
            }
        }
        if (last_visited != null) this.splay(last_visited);
        return res;
    }
    findKthNode(rank) {
        if (rank < 0 || rank >= this.size()) return null;
        let node = this.root;
        while (node != null) {
            let leftsize = node.left == null ? 0 : node.left.sz;
            if (leftsize == rank) break;
            if (leftsize > rank) {
                node = node.left;
            } else {
                rank -= leftsize + 1;
                node = node.right;
            }
        }
        this.splay(node);
        return node;
    }
    make(value) {
        return new SplayNode(value);
    }
    removeNode(node) {
        node = null;
    }

    // -------------------------------- Public Usage ----------------------------------------
    insert(value) { // allow duplicates  LST.set()
        if (this.root == null) {
            this.root = this.make(value);
            return this.root;
        }
        let node = this.root;
        while (node != null) {
            if (this.cmp(value, node.val)) {
                if (node.left == null) {
                    node.left = this.make(value);
                    node.left.parent = node;
                    node = node.left;
                    break;
                }
                node = node.left;
            } else {
                if (node.right == null) {
                    node.right = this.make(value);
                    node.right.parent = node;
                    node = node.right;
                    break;
                }
                node = node.right;
            }
        }
        this.splay(node);
        return node;
    }
```

```
    remove(value) { // remove one node, not all   LST.unset()
        let node = this.find(value);
        if (node == null) return false;
        this.splay(node);
        if (node.left == null) {
            this.root = node.right;
            if (node.right != null) node.right.parent = null;
            this.removeNode(node);
            return true;
        }
        if (node.right == null) {
            this.root = node.left;
            if (node.left != null) node.left.parent = null;
            this.removeNode(node);
            return true;
        }
        let last_node = this.LastNode(node);
        let next_node = this.NextNode(node);
        this.splay(last_node);
        this.splay(next_node, last_node);
        this.removeNode(next_node.left);
        next_node.left = null;
        next_node.update();
        last_node.update();
        return true;
    }
    has(value) { // LST.get()
        return this.count(value) > 0;
    }
    count(value) {
        let x = this.findFirstOf(value);
        if (x == null) return 0;
        let rank_x = this.findRankOf(x);
        let y = this.findLastOf(value);
        let rank_y = this.findRankOf(y);
        return rank_y - rank_x + 1;
    }
    rankOf(value) { // The number of elements strictly less than value
        let x = this.findPrecursorOf(value);
        return x == null ? 0 : this.findRankOf(x) + 1;
    }
    findKth(rank) { // (0-indexed)
        let x = this.findKthNode(rank);
        return x == null ? null : (x.val);
    }
    higher(value) { // > upper_bound()  LST.next(value)
        let node = this.findSuccessorOf(value);
        return node == null ? null : (node.val);
    }
    lower(value) { // <  LST.prev(value - 1)
        let node = this.findPrecursorOf(value);
        return node == null ? null : (node.val);
    }
    ceiling(value) { // >=
        return this.has(value) ? value : this.higher(value);
    }
    floor(value) { // <=
        return this.has(value) ? value : this.lower(value);
    }
    first() {
        return this.findKth(0);
    }
    last() {
        return this.findKth(this.size() - 1);
    }
    poll() {
        let res = this.first();
        this.remove(res);
        return res;
    }
    pollLast() {
        let res = this.last();
        this.remove(res);
        return res;
    }
```

```
    size() {
        return this.root == null ? 0 : this.root.sz;
    }
    isEmpty() {
        return this.root == null;
    }
    show() {
        let res = [];
        const dfs = (x) => {
            if (x == null) return;
            dfs(x.left);
            res.push(x.val);
            dfs(x.right);
        };
        dfs(this.root);
        return res;
    }
}

const N = 32;
const checkIthBit = (x, i) => x & (1 << i);

const minOperations = (a, t) => {
    let sum = 0, smaller = 0, tree = new SplayTree(), res = 0;
    a.map(x => {
        sum += x;
        tree.insert(x);
    });
    if (t > sum) return -1;
    for (let i = 0; i < N; i++) {
        let v = 1 << i;
        smaller += tree.count(v) << i;
        if (checkIthBit(t, i)) {
            if (smaller >= v) {
                smaller -= v;
            } else {
                for (let j = i + 1; j < N; j++) {
                    let vj = 1 << j;
                    if (tree.has(vj)) {
                        tree.remove(vj);
                        smaller += vj;
                        smaller -= v;
                        res += j - i;
                        break;
                    }
                }
            }
        }
    }
    return res;
};
```

Custom Testcase | Use Example Testcases

Run | Submit

Submission Result: Accepted (/submissions/detail/1032848456/) | More Details > (/submissions/detail/1032848456/)

Share your acceptance!