<https://leetcode.com/problems/fair-distribution-of-cookies/description/>

You are given an integer array cookies, where cookies[i] denotes the number of cookies in the ith bag. You are also given an integer k that denotes the number of children to distribute **all** the bags of cookies to. All the cookies in the same bag must go to the same child and cannot be split up.

The **unfairness** of a distribution is defined as the **maximum** **total** cookies obtained by a single child in the distribution.

Return *the****minimum****unfairness of all distributions*.

**Example 1:**

**Input:** cookies = [8,15,10,20,8], k = 2

**Output:** 31

**Explanation:** One optimal distribution is [8,15,8] and [10,20]

- The 1st child receives [8,15,8] which has a total of 8 + 15 + 8 = 31 cookies.

- The 2nd child receives [10,20] which has a total of 10 + 20 = 30 cookies.

The unfairness of the distribution is max(31,30) = 31.It can be shown that there is no distribution with an unfairness less than 31.

**Example 2:**

**Input:** cookies = [6,1,3,2,2,4,1,2], k = 3

**Output:** 7

**Explanation:** One optimal distribution is [6,1], [3,2,2], and [4,1,2]

- The 1st child receives [6,1] which has a total of 6 + 1 = 7 cookies.

- The 2nd child receives [3,2,2] which has a total of 3 + 2 + 2 = 7 cookies.

- The 3rd child receives [4,1,2] which has a total of 4 + 1 + 2 = 7 cookies.

The unfairness of the distribution is max(7,7,7) = 7.It can be shown that there is no distribution with an unfairness less than 7.

**Constraints:**

2 <= cookies.length <= 8

1 <= cookies[i] <= 105

2 <= k <= cookies.length

**Attempt 1: 2025-08-28**

**Solution 1: Binary Search + Greedy (30 min)**

**Differences:**

**410. Split Array Largest Sum**: Arrays must be contiguous subarrays

**2305. Fair Distribution of Cookies**: Can assign any cookie to any child (non-contiguous)

**410** is generally easier as it deals with contiguous segments

**2305** requires more sophisticated backtracking due to non-contiguous assignment

class Solution {

    public int distributeCookies(int[] cookies, int k) {

        int lo = 0;

        int hi = 0;

        for(int cookie : cookies) {

            lo = Math.max(cookie, lo);

            hi += cookie;

        }

        while(lo <= hi) {

            int mid = lo + (hi - lo) / 2;

            if(canDistribute(cookies, mid, k)) {

                hi = mid - 1;

            } else {

                lo = mid + 1;

            }

        }

        return lo;

    }

    // Why use backtrack in L2305 but not in L410 ?

    // - 410. Split Array Largest Sum: Arrays must be contiguous subarrays

    // - 2305. Fair Distribution of Cookies: Can assign any cookie to any child (non-contiguous)

    // - 410 is generally easier as it deals with contiguous segments

    // - 2305 requires more sophisticated backtracking due to non-contiguous assignment

    private boolean canDistribute(int[] cookies, int unfairness, int k) {

        return helper(cookies, new int[k], unfairness, 0);

    }

    private boolean helper(int[] cookies, int[] children, int unfairness, int index) {

        if(index == cookies.length) {

            return true;

        }

        for(int i = 0; i < children.length; i++) {

            if(children[i] + cookies[index] > unfairness) {

                continue;

            }

            if(i > 0 && children[i] == children[i - 1]) {

                continue;

            }

            children[i] += cookies[index];

            if(helper(cookies, children, unfairness, index + 1)) {

                return true;

            }

            children[i] -= cookies[index];

        }

        return false;

    }

}

Time Complexity: O(n \* log(S) \* k^n)

Where S is the total sum of all cookies.

Breakdown:

Binary search: O(log(S)) iterations (S = total cookie sum)

Each validation: O(k^n) in worst case for backtracking

Total: O(log(S) \* k^n)

More precise analysis:

Binary search range: from max(cookie) to sum(cookies) = O(log(S))

Each validation uses backtracking with pruning: O(k^n) worst-case

With pruning, actual validation time is much better in practice

Space Complexity: O(n + k)

Binary search: O(1) additional space

Validation backtracking: O(n) recursion stack + O(k) for children array

Total: O(n + k)

**Solution 2: Backtracking + Sorting (30 min)**

class Solution {

    private int minUnfairness = Integer.MAX\_VALUE;

    public int distributeCookies(int[] cookies, int k) {

        int[] children = new int[k];

        helper(cookies, 0, children);

        return minUnfairness;

    }

    private void helper(int[] cookies, int index, int[] children) {

        if(index == cookies.length) {

            int max = 0;

            for(int amount : children) {

                max = Math.max(max, amount);

            }

            minUnfairness = Math.min(minUnfairness, max);

            return;

        }

        // Prune: if current max is already worse than best found, return

        int currentMax = 0;

        for(int amount : children) {

            currentMax = Math.max(currentMax, amount);

        }

        if(currentMax >= minUnfairness) {

            return;

        }

        for(int i = 0; i < children.length; i++) {

            children[i] += cookies[index];

            helper(cookies, index + 1, children);

            children[i] -= cookies[index];

        }

    }

}

Time Complexity: O(k^n)

Worst-case: Without pruning, it would be O(k^n) where n is number of cookies and k is number of children

With pruning: Significantly reduced in practice, but still exponential in worst case

Best-case: With optimal pruning, it can approach O(k^(n/k)) in some cases

Explanation:

Each cookie can be assigned to any of k children

Without pruning: k choices for each of n cookies = k^n

With pruning: We skip duplicate states and cut off branches early

Space Complexity: O(n + k)

Recursion stack: O(n) depth (one level per cookie)

Children array: O(k) to store current distribution

Total: O(n + k)

**Refer to Deepseek**

**Approach**

This problem is about distributing cookies among k children such that the maximum total cookies obtained by any child is minimized. It's similar to:

**LeetCode 410. Split Array Largest Sum** (minimize maximum subarray sum)

**LeetCode 698. Partition to K Equal Sum Subsets** (partition into k equal subsets)

**LeetCode 1986. Minimum Number of Work Sessions** (similar session-based distribution)

We can use **backtracking with pruning** or **binary search with greedy checking**.

**Solution 1: Backtracking with Pruning (Recommended)**

class Solution {

    private int minUnfairness = Integer.MAX\_VALUE; // Tracks the minimum maximum cookies any child gets

    public int distributeCookies(int[] cookies, int k) {

        int[] children = new int[k]; // Array to track cookie count for each child

        backtrack(cookies, 0, children); // Start backtracking from first cookie

        return minUnfairness;

    }

    private void backtrack(int[] cookies, int index, int[] children) {

        // Base case: all cookies have been distributed

        if (index == cookies.length) {

            int max = 0;

            // Find the maximum cookies any child has (the unfairness metric)

            for (int amount : children) {

                max = Math.max(max, amount);

            }

            // Update the global minimum unfairness

            minUnfairness = Math.min(minUnfairness, max);

            return;

        }

        // PRUNING: Calculate current maximum before distributing more cookies

        int currentMax = 0;

        for (int amount : children) {

            currentMax = Math.max(currentMax, amount);

        }

        // If current distribution is already worse than best found solution, stop exploring

        if (currentMax >= minUnfairness) {

            return;

        }

        // Try giving the current cookie to each child

        for (int i = 0; i < children.length; i++) {

            children[i] += cookies[index]; // Give cookie to this child

            backtrack(cookies, index + 1, children); // Recurse for next cookie

            children[i] -= cookies[index]; // Backtrack: take cookie back

            // IMPORTANT PRUNING: If this child had 0 cookies before receiving this cookie,

            // and we're backtracking (meaning this assignment didn't lead to a better solution),

            // then no need to try giving this cookie to other children who also have 0 cookies.

            // This avoids duplicate states where the same cookie is given to different empty children.

            if (children[i] == 0) {

                break;

            }

        }

    }

}

**Can you share an example for below prune to illustrate the logic ?**

// IMPORTANT PRUNING: If this child had 0 cookies before receiving this cookie,

// and we're backtracking (meaning this assignment didn't lead to a better solution),

// then no need to try giving this cookie to other children who also have 0 cookies.

// This avoids duplicate states where the same cookie is given to different empty children.

if (children[i] == 0) {

break;

}

Let me illustrate this critical pruning with a concrete example:

**Example Scenario**

**Cookies:** [8, 15, 10] (3 cookies)

**Children:** k = 3 (Child A, Child B, Child C)

**Current state:** All children have 0 cookies initially

**Without Pruning**

We would try all permutations:

Give cookie#1 (8) to Child A → then try cookie#2 and cookie#3

Give cookie#1 (8) to Child B → then try cookie#2 and cookie#3

Give cookie#1 (8) to Child C → then try cookie#2 and cookie#3

But all these are **equivalent states** since the children are indistinguishable when empty!

**With Pruning**

if (children[i] == 0) {

break;

}

Here's what happens:

**First iteration:** Give cookie#1 (8) to Child A

children = [8, 0, 0]

Recurse to distribute remaining cookies

Backtrack: Child A goes back to 0

**Second iteration:** Try to give cookie#1 (8) to Child B

But children[i] == 0 is true (Child B has 0 cookies)

break out of the loop ← **PRUNING HAPPENS HERE**

Skip trying Child C entirely

**Why This Works**

**Mathematically equivalent states:**

[8, 0, 0] (cookie to Child A)

[0, 8, 0] (cookie to Child B)

[0, 0, 8] (cookie to Child C)

These all represent the **same distribution pattern**: one child has 8 cookies, others have 0.

**More Complex Example**

**Cookies:** [8, 15, 10, 5]

**Partial distribution:** [8, 15, 0, 0] (Child A=8, Child B=15, Child C=0, Child D=0)

Now we want to distribute cookie#3 (10):

Option 1: Give to Child C → [8, 15, 10, 0]

Option 2: Give to Child D → [8, 15, 0, 10] ← EQUIVALENT to Option 1!

With pruning:

Give to Child C → explore this branch fully

When backtracking, Child C goes back to 0

Try to give to Child D → but children[i] == 0 → break

**The Logic Behind the Pruning**

for (int i = 0; i < children.length; i++) {

children[i] += cookies[index];

backtrack(...);

children[i] -= cookies[index];

if (children[i] == 0) { // After backtracking, if child is empty again

break; // No need to try other empty children

}

}

**Key insight: If giving a cookie to an empty child didn't lead to a better solution, then giving it to any other empty child won't either, because empty children are indistinguishable.**

This pruning eliminates **symmetrical duplicate states** and reduces the time complexity from O(k^n) to roughly O(k^(n/k)) in practice, making it feasible for the problem constraints.

**Solution 2: Binary Search + Greedy (Similar to LeetCode 410)**

class Solution {

public int distributeCookies(int[] cookies, int k) {

int left = 0;

int right = 0;

for (int cookie : cookies) {

left = Math.max(left, cookie);

right += cookie;

}

while (left < right) {

int mid = left + (right - left) / 2;

if (canDistribute(cookies, k, mid)) {

right = mid;

} else {

left = mid + 1;

}

}

return left;

}

private boolean canDistribute(int[] cookies, int k, int maxLimit) {

return backtrack(cookies, 0, new int[k], maxLimit);

}

private boolean backtrack(int[] cookies, int index, int[] children, int maxLimit) {

if (index == cookies.length) {

return true;

}

for (int i = 0; i < children.length; i++) {

if (children[i] + cookies[index] > maxLimit) {

continue;

}

children[i] += cookies[index];

if (backtrack(cookies, index + 1, children, maxLimit)) {

return true;

}

children[i] -= cookies[index];

// Prune: if this child hasn't received any cookies yet,

// no need to try other empty children

if (children[i] == 0) {

break;

}

}

return false;

}

}

**Solution 3: Optimized Backtracking with Sorting**

class Solution {

private int minUnfairness = Integer.MAX\_VALUE;

public int distributeCookies(int[] cookies, int k) {

// Sort in descending order for better pruning

Arrays.sort(cookies);

reverse(cookies);

int[] children = new int[k];

backtrack(cookies, 0, children);

return minUnfairness;

}

private void backtrack(int[] cookies, int index, int[] children) {

if (index == cookies.length) {

int max = 0;

for (int amount : children) {

max = Math.max(max, amount);

}

minUnfairness = Math.min(minUnfairness, max);

return;

}

// Prune early if current max is already worse

int currentMax = getCurrentMax(children);

if (currentMax >= minUnfairness) {

return;

}

for (int i = 0; i < children.length; i++) {

// Skip duplicate states: if previous children have same amount,

// no need to try this child (produces same result)

if (i > 0 && children[i] == children[i - 1]) {

continue;

}

children[i] += cookies[index];

backtrack(cookies, index + 1, children);

children[i] -= cookies[index];

// Prune: if this child was empty, no need to try other empty children

if (children[i] == 0) {

break;

}

}

}

private int getCurrentMax(int[] children) {

int max = 0;

for (int amount : children) {

max = Math.max(max, amount);

}

return max;

}

private void reverse(int[] arr) {

int left = 0, right = arr.length - 1;

while (left < right) {

int temp = arr[left];

arr[left] = arr[right];

arr[right] = temp;

left++;

right--;

}

}

}

**Comparison with LeetCode 410**

**Similarities:**

Both aim to minimize the maximum sum in a partition

Both can use binary search + greedy validation

Both can use backtracking with pruning

**Differences:**

**410. Split Array Largest Sum**: Arrays must be contiguous subarrays

**2305. Fair Distribution of Cookies**: Can assign any cookie to any child (non-contiguous)

**410** is generally easier as it deals with contiguous segments

**2305** requires more sophisticated backtracking due to non-contiguous assignment

**Recommendation**

The **backtracking with pruning** (Solution 1) is recommended for this problem because:

The constraints are small (cookies.length <= 8, k <= 8)

It's straightforward and easy to understand

The pruning makes it efficient enough for the problem size

The binary search approach (Solution 2) is more suitable for larger constraints, but for this problem size, backtracking is sufficient and simpler.

**What's the time and space complexity for these 2 solutions ?**

**Solution 1: Backtracking with Pruning**

**Time Complexity: O(k^n)**

**Worst-case**: Without pruning, it would be O(k^n) where n is number of cookies and k is number of children

**With pruning**: Significantly reduced in practice, but still exponential in worst case

**Best-case**: With optimal pruning, it can approach O(k^(n/k)) in some cases

**Explanation**:

Each cookie can be assigned to any of k children

Without pruning: k choices for each of n cookies = k^n

With pruning: We skip duplicate states and cut off branches early

**Space Complexity: O(n + k)**

**Recursion stack**: O(n) depth (one level per cookie)

**Children array**: O(k) to store current distribution

**Total**: O(n + k)

**Solution 2: Binary Search + Greedy**

**Time Complexity: O(n \* log(S) \* k^n)**

Where S is the total sum of all cookies.

**Breakdown:**

**Binary search**: O(log(S)) iterations (S = total cookie sum)

**Each validation**: O(k^n) in worst case for backtracking

**Total**: O(log(S) \* k^n)

**More precise analysis:**

Binary search range: from max(cookie) to sum(cookies) = O(log(S))

Each validation uses backtracking with pruning: O(k^n) worst-case

With pruning, actual validation time is much better in practice

**Space Complexity: O(n + k)**

**Binary search**: O(1) additional space

**Validation backtracking**: O(n) recursion stack + O(k) for children array

**Total**: O(n + k)

**Solution 3: Optimized Backtracking with Sorting**

**Time Complexity: O(k^n) but with better constants**

**Sorting**: O(n log n) - negligible compared to backtracking

**Backtracking**: Still O(k^n) worst-case, but much better pruning:

Descending sort: larger cookies first → better pruning

Duplicate state skipping: avoids redundant computations

Empty child pruning: reduces branching factor

**Space Complexity: O(n + k)**

Same as Solution 1

**Practical Performance Comparison**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Solution | Worst-case Time | Average Time | Space | Recommended |
| 1 (Basic Backtracking) | O(k^n) | Moderate | O(n+k) | Good for small n |
| 2 (Binary Search) | O(log(S) \* k^n) | Slower | O(n+k) | Not recommended for this problem |
| 3 (Optimized Backtracking) | O(k^n) | Fastest | O(n+k) | **Best choice** |

**Why Solution 3 is Best for This Problem**

**Constraints are small**: n ≤ 8, k ≤ 8

8^8 = 16,777,216 operations worst-case

With pruning, actual operations are much fewer

**Pruning is very effective**:

Descending sort: fills larger cookies first, triggering constraints early

Duplicate state skipping: avoids trying identical distributions

Empty child pruning: prevents redundant assignments to empty children

**Binary search overhead**: For small n, the binary search doesn't help much and adds log(S) factor

**Example with n=8, k=8:**

**Without pruning**: 8^8 = 16.7 million operations

**With pruning**: Typically < 100,000 operations in practice

**Binary search**: Would do ~log2(10^6) ≈ 20 iterations × 100,000 = 2 million operations

For this problem size, the optimized backtracking (Solution 3) is the most efficient approach.

**Refer to**

[L410.Split Array Largest Sum (Ref.L1011,L704,L1482)](note://WEB4d9d6f26a965381b137615128e2648be)

[L473.Matchsticks to Square (Ref.L2397)](note://WEB803bb8fa17c70b8a004a7117d62b41b1)

[L698.Partition to K Equal Sum Subsets (Ref.L473,L2397)](note://WEBddd6d987abe5a5b2625172e67e298089)

[L1986.Minimum Number of Work Sessions to Finish the Tasks (Ref.L473,L698)](note://WEB89175f2f8ba2facb8115b472227a272b)

[L2397.Maximum Rows Covered by Columns (Ref.L473)](note://WEB050bb59c3f216a8c91af09746dbf6efe)