Elevator Rides

There are n people who want to get to the top of a building which has only one elevator.
You know the weight of each person and the maximum allowed weight in the elevator.

What is the minimum number of elevator rides?

Input

The first input line has two integers n and x: the number of people and the maximum allowed weight in the elevator.

The second line has n integers w_1,w_2,..,w_n: the weight of each person.

Output

Print one integer: the minimum number of rides.

Constraints

1 <= n <= 20
1 <= x <= 10^9
1 <= w_i <= x

Example

Input:

4 10
4 8 6 1

Output:

2

Problem Breakdown

Lets understand the problem first

We want the minimum number of elevator rides needed to take all people up, given:

- Each person has a weight w[i]
- Elevator has max capacity x
- n ≤ 20

DP State

For each subset of people mask, we want to know:

What is the best (minimum) way to take exactly those people using some number of rides?

But just knowing the number of rides is not enough.

Example:

- Maybe we need 3 rides, and the last ride has 2 people with total weight 7.
- Or 3 rides, but the last ride has total weight 2.

Between these two, we prefer the one with last ride weight 2, because it has more "space" to potentially add more people later.

So for each subset mask, we store:

dp[mask] = (rides, last_weight)

where:

- rides = minimum number of rides needed for this subset,
- last_weight = total weight of people in the current (last) ride for this subset.

The entire setup is like saying:

Among all ways to group these people into rides, we pick the way that:

1. Uses the fewest rides.
2. If several ways use the same number of rides, choose the one with the smallest last_weight.

That second criterion gives us more flexibility to add more people later.

Base case

When mask = 0 (no people taken):

We can think we've "started" the first ride, but it has weight 0.

So:

dp[0] = (1, 0)

Meaning: we are on ride #1, and it currently has weight 0.

You can also think of it as:
"We're ready for the first ride."

Transition

We want to build larger subsets from smaller ones.

For a given mask, we know dp[mask] = (rides, last_weight) is optimal for that subset.

Now we try to add one more person i who is not in mask:

- That means bit i is 0 in mask.
- New subset will be newMask = mask | (1 << i).

We have two cases:

Case 1: Person fits in the current (last) ride

If last_weight + w[i] <= x:

We can put person i into the current ride.

New state:

candidate = (rides, last_weight + w[i])

Case 2: Person does not fit in the current ride

If last_weight + w[i] > x:

We must start a new ride for this person.

New state:

candidate = (rides + 1, w[i])

Code

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int n;
    long long x;
    cin >> n >> x;

    vector<long long> w(n);
    for (int i = 0; i < n; i++) {
        cin >> w[i];
    }

    int N = 1 << n; // total number of subsets

    // dp[mask] = {minimum number of rides, weight of last ride}
    const long long INF = (long long)1e18;
    vector<pair<long long, long long>> dp(N, {INF, INF});

    // base case: no one taken yet, we are "on" the first ride with weight 0
    dp[0] = {1, 0};

    for (int mask = 0; mask < N; mask++) {
        auto [rides, last_w] = dp[mask];
        if (rides == INF) continue; // unreachable state, skip

        // try to add each person i who is not in 'mask'
        for (int i = 0; i < n; i++) {
            if (mask & (1 << i)) continue; // person i already included

            int newMask = mask | (1 << i);
            pair<long long, long long> candidate;

            if (last_w + w[i] <= x) {
                // put this person in the current (last) ride
                candidate = {rides, last_w + w[i]};
            } else {
                // start a new ride with this person
                candidate = {rides + 1, w[i]};
            }

            // keep the better (min) option for dp[newMask]
            dp[newMask] = min(dp[newMask], candidate);
        }
    }

    int fullMask = N - 1;
    cout << dp[fullMask].first << "\n";

    return 0;
}
```