<https://leetcode.com/problems/minimum-number-of-work-sessions-to-finish-the-tasks/description/>

There are n tasks assigned to you. The task times are represented as an integer array tasks of length n, where the ith task takes tasks[i] hours to finish. A **work session** is when you work for **at most** sessionTime consecutive hours and then take a break.

You should finish the given tasks in a way that satisfies the following conditions:

If you start a task in a work session, you must complete it in the **same** work session.

You can start a new task **immediately** after finishing the previous one.

You may complete the tasks in **any order**.

Given tasks and sessionTime, return *the****minimum****number of****work sessions****needed to finish all the tasks following the conditions above.*

The tests are generated such that sessionTime is **greater** than or **equal** to the **maximum** element in tasks[i].

**Example 1:**

**Input:** tasks = [1,2,3], sessionTime = 3]

**Output:** 2

**Explanation:** You can finish the tasks in two work sessions.

- First work session: finish the first and the second tasks in 1 + 2 = 3 hours.

- Second work session: finish the third task in 3 hours.

**Example 2:**

**Input:** tasks = [3,1,3,1,1], sessionTime = 8

**Output:** 2

**Explanation:** You can finish the tasks in two work sessions.

- First work session: finish all the tasks except the last one in 3 + 1 + 3 + 1 = 8 hours.

- Second work session: finish the last task in 1 hour.

**Example 3:**

**Input:** tasks = [1,2,3,4,5], sessionTime = 15

**Output:** 1

**Explanation:** You can finish all the tasks in one work session.

**Constraints:**

n == tasks.length

1 <= n <= 14

1 <= tasks[i] <= 10

max(tasks[i]) <= sessionTime <= 15

**Attempt 1: 2025-08-27**

**Solution 1: Backtracking + Sorting (30 min, TLE)**

class Solution {

    int minSessions = Integer.MAX\_VALUE;

    public int minSessions(int[] tasks, int sessionTime) {

        Arrays.sort(tasks);

        for(int i = 0; i < tasks.length / 2; i++) {

            int tmp = tasks[i];

            tasks[i] = tasks[tasks.length - 1 - i];

            tasks[tasks.length - 1 - i] = tmp;

        }

        List<Integer> sessions = new ArrayList<>();

        helper(tasks, sessionTime, 0, sessions);

        return minSessions;

    }

    private void helper(int[] tasks, int sessionTime, int index, List<Integer> sessions) {

        if(index == tasks.length) {

            minSessions = Math.min(minSessions, sessions.size());

            return;

        }

        // Try adding to existing sessions

        for(int i = 0; i < sessions.size(); i++) {

            if(sessions.get(i) + tasks[index] > sessionTime) {

                continue;

            }

            sessions.set(i, sessions.get(i) + tasks[index]);

            helper(tasks, sessionTime, index + 1, sessions);

            sessions.set(i, sessions.get(i) - tasks[index]);

        }

        // Try with new session

        sessions.add(tasks[index]);

        helper(tasks, sessionTime, index + 1, sessions);

        sessions.remove(sessions.size() - 1);

    }

}

Time Complexity: Backtracking: O(n!) in worst case

Space Complexity: O(2^n) for DP, O(n) for backtracking

**Optimization with critical prune:**

class Solution {

    int minSessions = Integer.MAX\_VALUE;

    public int minSessions(int[] tasks, int sessionTime) {

        Arrays.sort(tasks);

        for(int i = 0; i < tasks.length / 2; i++) {

            int tmp = tasks[i];

            tasks[i] = tasks[tasks.length - 1 - i];

            tasks[tasks.length - 1 - i] = tmp;

        }

        List<Integer> sessions = new ArrayList<>();

        helper(tasks, sessionTime, 0, sessions);

        return minSessions;

    }

    private void helper(int[] tasks, int sessionTime, int index, List<Integer> sessions) {

        // Prune if already exceeded current minimum

        if (sessions.size() >= minSessions) {

            return;

        }

        if(index == tasks.length) {

            minSessions = Math.min(minSessions, sessions.size());

            return;

        }

        // Try adding to existing sessions

        for(int i = 0; i < sessions.size(); i++) {

            if(sessions.get(i) + tasks[index] > sessionTime) {

                continue;

            }

            sessions.set(i, sessions.get(i) + tasks[index]);

            helper(tasks, sessionTime, index + 1, sessions);

            sessions.set(i, sessions.get(i) - tasks[index]);

        }

        // Try with new session

        sessions.add(tasks[index]);

        helper(tasks, sessionTime, index + 1, sessions);

        sessions.remove(sessions.size() - 1);

    }

}

Time Complexity: Backtracking: O(n!) in worst case, but heavily pruned

Space Complexity: O(2^n) for DP, O(n) for backtracking

**Optimization with multiple prunes:**

class Solution {

    int minSessions = Integer.MAX\_VALUE;

    public int minSessions(int[] tasks, int sessionTime) {

        Arrays.sort(tasks);

        for(int i = 0; i < tasks.length / 2; i++) {

            int tmp = tasks[i];

            tasks[i] = tasks[tasks.length - 1 - i];

            tasks[tasks.length - 1 - i] = tmp;

        }

        List<Integer> sessions = new ArrayList<>();

        helper(tasks, sessionTime, 0, sessions);

        return minSessions;

    }

    private void helper(int[] tasks, int sessionTime, int index, List<Integer> sessions) {

        // Critical prune if already exceeded current minimum

        if (sessions.size() >= minSessions) {

            return;

        }

        if(index == tasks.length) {

            // Since introduce critical prune, no need comparison again

            //minSessions = Math.min(minSessions, sessions.size());

            minSessions = sessions.size();

            return;

        }

        // Try adding to existing sessions

        for(int i = 0; i < sessions.size(); i++) {

            if(sessions.get(i) + tasks[index] > sessionTime) {

                continue;

            }

            // Skip sessions with same capacity (avoid duplicate states)

            if(i > 0 && sessions.get(i).equals(sessions.get(i - 1))) {

                continue;

            }

            sessions.set(i, sessions.get(i) + tasks[index]);

            helper(tasks, sessionTime, index + 1, sessions);

            sessions.set(i, sessions.get(i) - tasks[index]);

        }

        // Only try new session if we haven't found optimal solution yet

        if(sessions.size() + 1 < minSessions) {

            sessions.add(tasks[index]);

            helper(tasks, sessionTime, index + 1, sessions);

            sessions.remove(sessions.size() - 1);

        }

    }

}

Time Complexity: Backtracking: O(n!) in worst case, but heavily pruned

Space Complexity: O(2^n) for DP, O(n) for backtracking

**Refer to Deepseek**

**Approach**

This problem requires finding the minimum number of work sessions needed to complete all tasks, where each session has a maximum duration. We can use:

**Backtracking with State Caching**: Try assigning tasks to sessions, memoizing states to avoid recomputation

**Bitmask DP**: Represent task assignments as bitmasks for efficient state representation

**Greedy Assignment**: Sort tasks in descending order for better pruning

**Solution Code (Bitmask DP)**

import java.util.\*;

class Solution {

public int minSessions(int[] tasks, int sessionTime) {

int n = tasks.length;

// dp[mask] = minimum sessions needed for tasks represented by bitmask 'mask'

int[] dp = new int[1 << n];

Arrays.fill(dp, Integer.MAX\_VALUE);

dp[0] = 0;

// Precompute time for each subset of tasks

int[] time = new int[1 << n];

for (int mask = 0; mask < (1 << n); mask++) {

int total = 0;

for (int i = 0; i < n; i++) {

if ((mask & (1 << i)) != 0) {

total += tasks[i];

}

}

time[mask] = total;

}

for (int mask = 0; mask < (1 << n); mask++) {

if (dp[mask] == Integer.MAX\_VALUE) continue;

// Generate all subsets of unassigned tasks

int unassigned = ((1 << n) - 1) & ~mask;

for (int subset = unassigned; subset > 0; subset = (subset - 1) & unassigned) {

if (time[subset] <= sessionTime) {

int newMask = mask | subset;

dp[newMask] = Math.min(dp[newMask], dp[mask] + 1);

}

}

}

return dp[(1 << n) - 1];

}

}

**Alternative Solution (Backtracking with Memoization)**

class Solution {

public int minSessions(int[] tasks, int sessionTime) {

Arrays.sort(tasks);

reverse(tasks);

return dfs(tasks, 0, new ArrayList<Integer>(), sessionTime);

}

private int dfs(int[] tasks, int index, List<Integer> sessions, int sessionTime) {

if (index == tasks.length) {

return sessions.size();

}

int minSessions = Integer.MAX\_VALUE;

// Try adding task to existing sessions

for (int i = 0; i < sessions.size(); i++) {

if (sessions.get(i) + tasks[index] <= sessionTime) {

sessions.set(i, sessions.get(i) + tasks[index]);

minSessions = Math.min(minSessions, dfs(tasks, index + 1, sessions, sessionTime));

sessions.set(i, sessions.get(i) - tasks[index]);

}

}

// Try creating new session

sessions.add(tasks[index]);

minSessions = Math.min(minSessions, dfs(tasks, index + 1, sessions, sessionTime));

sessions.remove(sessions.size() - 1);

return minSessions;

}

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--;

}

}

}

**Optimized Backtracking with Pruning**

class Solution {

private int minSessions = Integer.MAX\_VALUE;

public int minSessions(int[] tasks, int sessionTime) {

Arrays.sort(tasks);

reverse(tasks);

List<Integer> sessions = new ArrayList<>();

backtrack(tasks, 0, sessions, sessionTime);

return minSessions;

}

private void backtrack(int[] tasks, int index, List<Integer> sessions, int sessionTime) {

if (sessions.size() >= minSessions) {

return; // Prune if already exceeded current minimum

}

if (index == tasks.length) {

minSessions = Math.min(minSessions, sessions.size());

return;

}

// Try adding to existing sessions

for (int i = 0; i < sessions.size(); i++) {

if (sessions.get(i) + tasks[index] <= sessionTime) {

sessions.set(i, sessions.get(i) + tasks[index]);

backtrack(tasks, index + 1, sessions, sessionTime);

sessions.set(i, sessions.get(i) - tasks[index]);

}

}

// Try new session

sessions.add(tasks[index]);

backtrack(tasks, index + 1, sessions, sessionTime);

sessions.remove(sessions.size() - 1);

}

private void reverse(int[] arr) {

for (int i = 0; i < arr.length / 2; i++) {

int temp = arr[i];

arr[i] = arr[arr.length - 1 - i];

arr[arr.length - 1 - i] = temp;

}

}

}

**Explanation**

**Bitmask DP Approach**:

Use dynamic programming with bitmask to represent task assignments

Precompute total time for each subset of tasks

For each state, try all valid subsets of unassigned tasks that fit in a session

Update DP table with minimum sessions needed

**Backtracking Approach**:

Sort tasks in descending order for better pruning

Try assigning tasks to existing sessions or create new sessions

Use pruning to avoid paths that exceed current minimum sessions

**Optimizations**:

Descending sort: place larger tasks first to fill sessions efficiently

Pruning: stop early if current sessions exceed known minimum

Memoization: cache states to avoid recomputation (in DP solution)

**Time Complexity**:

Bitmask DP: O(3^n) - each task has 3 states (assigned, unassigned, being considered)

Backtracking: O(n!) in worst case, but heavily pruned

**Space Complexity**: O(2^n) for DP, O(n) for backtracking

The bitmask DP solution is more efficient for larger inputs, while backtracking is simpler to implement and works well with pruning.

**Looks like for backtracking solution if without below statement it will timeout when input is**

**tasks = [1,1,1,1,1,1,1,1,1,1,1,1,1,1]**

**sessionTime = 14**

// Prune if already exceeded current minimum

if (sessions.size() >= minSessions) {

return;

}

You're absolutely right! That pruning condition is **critical** for preventing timeout in the backtracking solution. Let me explain why and provide the complete solution with this essential optimization.

**Why the Pruning is Necessary**

For the input tasks = [1,1,1,1,1,1,1,1,1,1,1,1,1,1] with sessionTime = 14:

There are 14 tasks, each taking 1 unit of time

Each session can hold up to 14 tasks

The optimal solution is 1 session (all tasks fit in one session)

**Without pruning**, the backtracking would explore:

All possible ways to distribute 14 tasks into sessions

This creates an exponential number of combinations

For 14 tasks, it could explore billions of possibilities

**With pruning**, as soon as we have more sessions than the current minimum (which starts at 1), we immediately stop that branch.

**Additional Optimizations**

We can add even more pruning to make it faster:

class Solution {

private int minSessions = Integer.MAX\_VALUE;

public int minSessions(int[] tasks, int sessionTime) {

Arrays.sort(tasks);

reverse(tasks);

List<Integer> sessions = new ArrayList<>();

backtrack(tasks, 0, sessions, sessionTime);

return minSessions;

}

// Multiple pruning conditions

private void backtrack(int[] tasks, int index, List<Integer> sessions, int sessionTime) {

// CRITICAL PRUNING: Stop if we already have more sessions than current minimum

if (sessions.size() >= minSessions) return; // Already worse than best

if (index == tasks.length) {

minSessions = sessions.size();

return;

}

// Try existing sessions - skip duplicates

for (int i = 0; i < sessions.size(); i++) {

// Skip sessions with same capacity (avoid duplicate states)

if (i > 0 && sessions.get(i).equals(sessions.get(i - 1))) continue;

if (sessions.get(i) + tasks[index] <= sessionTime) {

sessions.set(i, sessions.get(i) + tasks[index]);

backtrack(tasks, index + 1, sessions, sessionTime);

sessions.set(i, sessions.get(i) - tasks[index]);

}

}

// Only try new session if we haven't found optimal solution yet

if (sessions.size() + 1 < minSessions) {

sessions.add(tasks[index]);

backtrack(tasks, index + 1, sessions, sessionTime);

sessions.remove(sessions.size() - 1);

}

}

private void reverse(int[] arr) {

for (int i = 0; i < arr.length / 2; i++) {

int temp = arr[i];

arr[i] = arr[arr.length - 1 - i];

arr[arr.length - 1 - i] = temp;

}

}

}

**Why This Works**

**Descending Sort**: Larger tasks first means we fill sessions more efficiently and hit constraints earlier

**Critical Pruning**: if (sessions.size() >= minSessions) return prevents exploring hopeless paths

**Duplicate Session Skipping**: Avoids trying the same session capacity multiple times

**Early New Session Pruning**: Only creates new sessions if it might lead to a better solution

Without that critical pruning condition, the solution would timeout on cases with many small tasks that can be combined in many ways. With pruning, it quickly finds the optimal solution and cuts off inferior branches.

**Refer to**

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

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