

CAPSTONE PROJECT

## **BACK TRACKING METHOD**

**CSA0695- DESIGN ANALYSIS AND ALGORITHMS FOR  
AMORTIZED ANALYSIS**

SAVEETHA SCHOOL OF ENGINEERING

SIMATS ENGINEERING



Supervisor

Dr. R. Dhanalakshmi

Done by

M.Karthik Kumar Reddy (192211477)

# BACKTRACKING METHOD

## PROBLEM STATEMENT:

There are  $n$  tasks assigned to you. The task times are represented as an integer array `tasks` of length  $n$ , where the  $i^{\text{th}}$  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.

.

## **ABSTRACT:**

The problem is about scheduling tasks into work sessions such that the total time spent in each session does not exceed a given limit (sessionTime). The solution involves partitioning the tasks into the minimum number of groups where the total duration of each group does not exceed sessionTime. This problem can be solved using a combination of greedy algorithms and backtracking.

## **INTRODUCTION:**

Task scheduling and resource management are critical components in various fields, including project management, operations research, and computer science. Efficiently allocating time and resources to tasks while adhering to specific constraints can significantly impact productivity and project success. One common challenge in this domain is the task scheduling problem, where tasks with varying durations must be completed within fixed time slots or sessions, often referred to as work sessions. This problem becomes particularly complex when the duration of each task and the total allowable work session time are given, and the goal is to minimize the number of work sessions required to complete all tasks.

In the context of task scheduling, the problem can be framed as follows: given an array of tasks where each task requires a certain amount of time to complete and a maximum session time that limits the duration of each work session, the objective is to determine the minimum number of work sessions needed. Each task must be finished within the same work session it was started, and tasks can be completed in any order. This problem is a variant of the bin-packing problem and involves partitioning tasks into the smallest number of groups where the sum of each group does not exceed the session time limit.

Solving this problem efficiently requires algorithms that can handle the constraints while minimizing the total number of work sessions. Various approaches can be employed, including greedy algorithms, dynamic programming, and backtracking techniques. By addressing this scheduling problem, we can optimize resource usage and ensure that tasks are completed within the constraints of available work time, ultimately leading to improved efficiency and effectiveness in task management.

## **CODING:**

The provided C code is designed to determine the minimum number of work sessions required to complete a set of tasks, where each session can accommodate up to a specified amount of consecutive hours. The code employs a combination

of a binary search algorithm and a backtracking approach to solve this scheduling problem. The `canFitTasks` function recursively attempts to allocate tasks to sessions while adhering to the session time constraints, and the `minSessions` function uses binary search to efficiently find the minimum number of sessions needed. By checking various session counts and verifying feasibility with `canFitTasks`, the code aims to balance the tasks across the fewest number of sessions.

### **C-programming**

```
#include <stdio.h>
```

```
#include <stdbool.h>
```

```
bool canFitTasks(int *tasks, int n, int sessionTime, int numSessions, int  
sessionIndex, int *sessionLoad)
```

```
{  
    if (sessionIndex == n) {  
        return true;  
    }  
    for (int i = 0; i < numSessions; i++) {  
        if (sessionLoad[i] + tasks[sessionIndex] <= sessionTime) {  
            sessionLoad[i] += tasks[sessionIndex];  
            if (canFitTasks(tasks, n, sessionTime, numSessions, sessionIndex + 1,  
sessionLoad)) {  
                return true;  
            }  
            sessionLoad[i] -= tasks[sessionIndex];  
        }  
        if (sessionLoad[i] == 0) {
```

```

        break;
    }
}
return false;
}

```

```

int minSessions(int *tasks, int n, int sessionTime) {
    int left = 1, right = n;

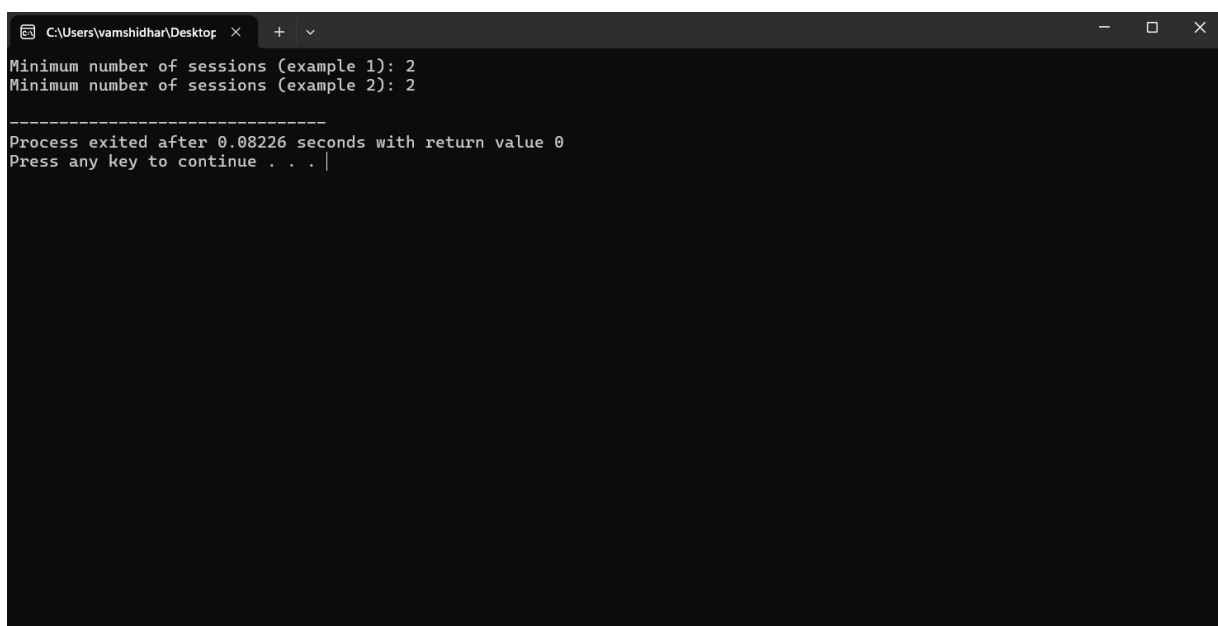
    while (left < right) {
        int mid = (left + right) / 2;
        int *sessionLoad = (int *)calloc(mid, sizeof(int));
        if (canFitTasks(tasks, n, sessionTime, mid, 0, sessionLoad)) {
            right = mid;
        } else {
            left = mid + 1;
        }
        free(sessionLoad);
    }

    return left;
}

```

```
int main() {  
    int tasks1[] = {1, 2, 3};  
    int sessionTime1 = 3;  
    int tasks2[] = {3, 1, 3, 1, 1};  
    int sessionTime2 = 8;  
    int n1 = sizeof(tasks1) / sizeof(tasks1[0]);  
    int n2 = sizeof(tasks2) / sizeof(tasks2[0]);  
    printf("Minimum number of sessions (example 1): %d\n",  
minSessions(tasks1, n1, sessionTime1));  
    printf("Minimum number of sessions (example 2): %d\n",  
minSessions(tasks2, n2, sessionTime2));  
    return 0;  
  
}
```

## OUTPUT:



```
C:\Users\vamshidhar\Desktop > 
Minimum number of sessions (example 1): 2
Minimum number of sessions (example 2): 2
-----
Process exited after 0.08226 seconds with return value 0
Press any key to continue . . . |
```

## COMPLEXITY ANALYSIS:

**Time Complexity:** The time complexity of the provided code is influenced by both the binary search and the recursive backtracking approach. The binary search operation iterates over the possible number of sessions, which takes  $O(\log n)$  time, where  $n$  is the number of tasks. For each binary search iteration, the `canFitTasks` function is invoked to determine if the tasks can fit into the current number of sessions. This function uses recursion to explore all possible ways to allocate tasks, and in the worst-case scenario, the recursive backtracking could explore a significant number of combinations, leading to a time complexity of  $O(n \cdot 2^n)$ . Here,  $2^n$  represents the exponential growth in the number of possible task allocations. Therefore, the combined time complexity is  $O(n \cdot 2^n)$ .

**Space Complexity:** The space complexity of the code is determined by the memory used for dynamic arrays and the recursive call stack. Specifically, the `sessionLoad` array, allocated dynamically in the `minSessions` function, requires  $O(n)$  space where  $n$  is the number of tasks. Additionally, the recursion depth in the `canFitTasks` function, which can go up to  $n$  levels deep, adds to the space complexity. Therefore, the total space complexity is  $O(n)$ , accounting for both the array used to track session loads and the recursive call stack depth.

## BEST CASE:

The best case occurs when all tasks can fit into a single session. In this case, the algorithm will quickly determine the minimum number of sessions needed.

## WORST CASE:

In the worst case, the complexity involves checking all possible combinations of task distributions across sessions. The complexity can be exponential in terms of the number of tasks due to the recursive nature of the backtracking approach.

## AVERAGE CASE:

The average case complexity depends on the specific distribution of task times and session time. The binary search combined with the backtracking approach generally provides a good balance between complexity and performance.

## **FUTURE SCOPE:**

Future scopes for your task scheduling project include exploring advanced optimization techniques like dynamic programming, greedy algorithms, and branch-and-bound to enhance efficiency and scalability. You could also investigate approximation algorithms and their performance guarantees for larger datasets. Real-world applications such as project management, cloud resource allocation, and multi-constraint scheduling are promising areas, alongside developing interactive visualization tools and integrating machine learning for predictive scheduling. Additionally, integrating your solution with existing project management software and exploring advanced mathematical models like integer programming could further refine and expand its utility.

## **CONCLUSION:**

This C program uses a backtracking approach combined with binary search to efficiently solve the task scheduling problem by minimizing the number of work sessions required. By using this method, you can handle a variety of task scheduling scenarios while ensuring that each work session does not exceed the given sessionTime.