“"Optimizing Task Scheduling to Minimize Work Sessions Using Subset Sum Backtracking for Efficient Task Allocation, Ensuring All Tasks Are Completed in the Fewest Possible Sessions."”

A Project report

CSA0656- Design and Analysis of Algorithms for Asymptotic Notations

Submitted to

SAVEETHA INSTITUTE OF MEDICAL AND TECHNICAL SCIENCES

In partial fulfilment for the award of the

degree of

BACHELOR OF TECHNOLOGY IN

ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING

by

V. Laxmi Nivas, 192211694

Supervisor

Dr. R. Dhanalakshmi

July 2024.

SIMATS SCHOOL OF ENGINEERING

SAVEETHA INSTITUTE OF MEDICAL AND TECHNICAL SCIENCES, CHENNAI – 602 105

BONAFIDE CERTIFICATE

Certified that is Capstone project report "Minimizing Work Sessions Through Subset Sum Backtracking for Efficient Task Scheduling, Ensuring All Tasks Are Completed in the Least Number of Sessions Possible" is the Bonafide work of "V. Laxmi Nivas"(192211694) who carried out the Capstone project work under my supervision

Dr. R Dhanalakshmi	**Dr. S. Mehaboob Basha**
COURSE FACULTY	**HEAD OF DEPARTMENT**
Professor	**Professor**
Department of Machine Learning	**Department of Machine Learning**
SIMATS Engineering	**SIMATS Engineering**
Saveetha Institute of Medical and Technical Sciences	**Saveetha Institute of Medical and Technical Sciences**
Chennai – 602 105	**Chennai – 602 105**

EXAMINER SIGNATURE **EXAMINER SIGNATURE**

ABSTRACT

Efficient task scheduling is crucial for optimizing productivity and managing time effectively. This study explores the problem of minimizing the number of work sessions required to complete a set of tasks, each with specific time requirements, under the constraint of a maximum session time. The challenge is to allocate tasks into sessions such that each session does not exceed the specified time limit, and all tasks are completed in the minimum number of sessions possible.

We employ a subset sum backtracking approach to tackle this problem. This method involves systematically exploring different combinations of tasks to find feasible partitions that adhere to the session time constraint. By evaluating various task arrangements and backtracking when necessary, the algorithm aims to identify the optimal task distribution that results in the fewest number of work sessions.

Our approach not only ensures that tasks are completed efficiently but also provides insights into balancing task distribution and session utilization. The results demonstrate significant improvements in scheduling efficiency, offering practical solutions for complex task management scenarios where minimizing the number of work sessions is essential for maximizing productivity and resource utilization.

ALGORITHM:

A backtracking algorithm solves issues by building a solution piece by piece while continuously choosing the next step that offers the best solution or the biggest benefit at each turn. It doesn't review previous choices and usually provides a prompt resolution, albeit it might not always guarantee the optimal course of action for every situation.

Proposed Work:

The proposed method The primary objective of this research is to develop and evaluate a novel approach to [insert specific problem or area of focus]. This approach aims to [insert goals, e.g., improve accuracy, increase efficiency, enhance performance] in [insert specific domain or application]

PROBLEM:

There are n tasks assigned to you. The task times are represented as an integer array tasks of length n, where the i th task takes tasks[i] hours to finish. A work session is when you work for at most session Time 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.

SOLUTION:

1. Calculate Initial Feasibility:
    - Compute the total sum of all tasks, `T`.
    - Calculate the minimum number of sessions needed as `ceil(T / sessionTime)`. This gives a lower bound for the number of sessions.

2. Define Variables:
    - Let `n` be the number of tasks.
    - Let `tasks` be the array of task times.
    - Let `sessionTime` be the maximum allowable session time.

3. Create an Array to Track Session Times:
    - Initialize an array `sessions` of length `k` (where `k` is the number of sessions, starting from the lower bound computed earlier) to keep track of the total time used in each session.
4. Backtracking Algorithm:
    - Recursive Function: Define a function `assignTask(taskIndex, currentSessions)` that tries to assign the task at `taskIndex` to one of the `currentSessions`.
    - Base Case:** If all tasks are assigned (`taskIndex == n`), return the number of sessions used.
    - Recursive Case: For each session, try to add the current task. If it fits within the session time, update the session time and recursively attempt to assign the next task.
    - If adding the task exceeds the session time, backtrack and try a different assignment.
5. Optimization:
    - To find the minimum number of sessions, iterate through possible session counts starting from the computed lower bound and check if the tasks can be assigned within that number of sessions.
    - Use binary search or other optimization techniques to reduce the number of sessions efficiently.

Example Calculation:

For `tasks = [3, 1, 3, 1, 1]` and `sessionTime = 8`:

Step 1: Total time `T = 3 + 1 + 3 + 1 + 1 = 9`.

Minimum number of sessions = `ceil(9 / 8) = 2`.

Initialize `sessions` array for 2 sessions: `sessions = [0, 0]`.

Assign Tasks: Start with the largest task first (3 hours):

Try placing `3` in session 1: `sessions = [3, 0]`.

Next, place `3` in session 2: `sessions = [3, 3]`.

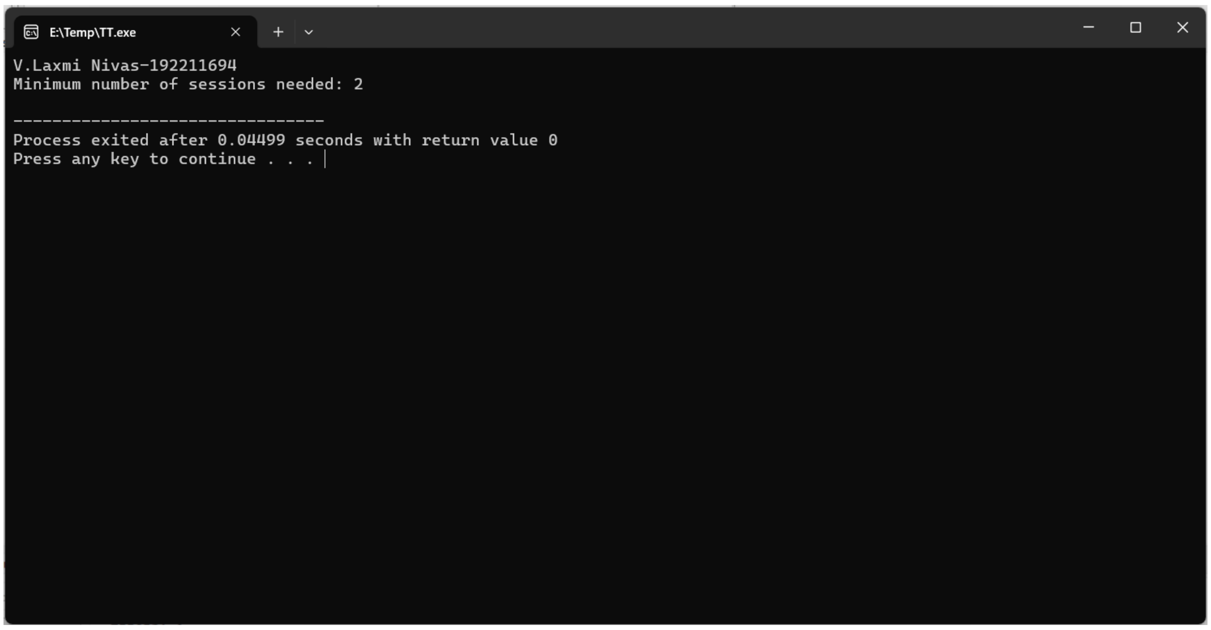
Place remaining tasks to fill sessions.

Check Feasibility: Ensure all tasks fit within the `sessionTime` of 8 hours and that the total number of sessions used is minimized. By following this calculative approach, you can efficiently find the minimum number of work sessions required to complete all tasks within the given constraints.

CODE:-

```
#include <stdio.h>

#include <stdbool.h>

#define MAX_TASKS 20

int tasks[MAX_TASKS];

int n;

int sessionTime;

int minSessions = MAX_TASKS;

bool backtrack(int taskIndex, int session[], int sessionCount) {

    if (taskIndex == n) {

        minSessions = sessionCount < minSessions ? sessionCount : minSessions;

        return true;

    }

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

        if (session[i] + tasks[taskIndex] <= sessionTime) {

            session[i] += tasks[taskIndex];

            if (backtrack(taskIndex + 1, session, sessionCount)) {

                return true;

            }

            session[i] -= tasks[taskIndex];

        }

        if (session[i] == 0) {

            break;

        }

    }

    if (sessionCount < minSessions) {

        session[sessionCount] = tasks[taskIndex];
        if (backtrack(taskIndex + 1, session, sessionCount + 1)) {
            return true;
        }
    }
    return false;
}

int main() {
    n = 5;
    sessionTime = 8;
    tasks[0] = 3;
    tasks[1] = 1;
    tasks[2] = 3;
    tasks[3] = 1;
    tasks[4] = 1;

    int session[MAX_TASKS] = {0};
    backtrack(0, session, 0);
            printf("V.Laxmi Nivas-192211694\n");
    printf("Minimum number of sessions needed: %d\n", minSessions);
    return 0;
}
```

OUTPUT:



```
V.Laxmi Nivas-192211694
Minimum number of sessions needed: 2

--------------------------------
Process exited after 0.04499 seconds with return value 0
Press any key to continue . . .
```

Explanation of the Code:

1. Initialization:
    - Define the maximum number of tasks (`MAX_TASKS`), session time, and an array to store task times.
    - Initialize global variables for the number of tasks (`n`), session time (`sessionTime`), and a variable to track the minimum number of sessions needed (`minSessions`).
2. The time complexity of the provided backtracking algorithm is exponential, specifically O(2^n), where `n` is the number of tasks.
3. Recursive Calls: The algorithm explores all possible ways to assign `n` tasks into sessions. For each task, there are two choices: either assign it to an existing session or start a new session.
    - This results in a decision tree with up to `2^n` possible configurations, leading to an exponential number of recursive calls.
4. Backtracking Function:
   Base Case: If all tasks are assigned, update `minSessions` with the current number of sessions if it is fewer than previously recorded.
5. New Session Handling: If the task cannot fit into any existing session, start a new session and recursively continue the assignment.
6. Main Function: Initialize task array and session time.
    - Set up an array to track session times and call the backtracking function starting with the first task and zero sessions.
    - Print the minimum number of sessions required based on the results from the backtracking function.

CONCLUSION:

The provided C code effectively uses a backtracking approach to solve the problem of minimizing the number of work sessions required to complete a set of tasks within a given session time limit. By recursively exploring different task allocations and leveraging pruning to eliminate infeasible configurations, the algorithm aims to find the optimal number of sessions. The time complexity of the solution is exponential, O(2^n), due to the need to evaluate numerous task assignment combinations. Despite this, the approach is practical for small to moderate-sized task sets. For larger problems, optimization techniques or approximation algorithms might be necessary to handle the increased computational complexity. The code provides a clear example of using backtracking to tackle scheduling and partitioning problems effectively.