MAXPROFIT

TABLE OF CONTENTS

1.Problem Statement

Mr. X owns a large strip of land on Mars. He can develop this land by constructing one of the following buildings:

1. Theatre
   a. Build time: 5 units
   b. Earning: $1500 per time unit while operational (after construction completes)
   c. Land size: 2 × 1
2. Pub
   a. Build time: 4 units
   b. Earning: $1000 per time unit while operational (after construction completes)
   c. Land size: 1 × 1
3. Commercial Park
   a. Build time: 10 units
   b. Earning: $3000 per time unit while operational (after construction completes)
   c. Land size: 3 × 1

Additional information:

- Land is unlimited, so land constraints do not apply.
- Total available development time is n units.
- Only one building can be under construction at any time (no parallel builds).
- Once a building finishes, it earns the stated amount **for each remaining time unit** until time n.
  - Concretely, if a building finishes at time t ($0 \leq t \leq n$), it contributes $(n - t) \times$ (earning per time unit) to the total profit.
  - If a building finishes exactly at $t = n$, it yields 0 because there are no remaining operational periods.
- The objective is to schedule builds (possibly multiple of the same type, sequentially) to **maximize total profit** within the n time units.

2.Why Unbounded Knapsack

The Max Profit Problem aligns closely with the characteristics of the unbounded knapsack problem in algorithm design. In this scenario, Mr. X has a fixed amount of time, which serves as the constrained resource,similar to the weight or capacity in the knapsack model. Each building type (Theatre, Pub, or Commercial Park) consumes a certain number of time units to develop and yields earnings per operational time unit after construction is completed.

Importantly, there is no restriction on the number of times a particular building can be constructed, which means the same choice can be made repeatedly,this is a defining feature of unbounded knapsack problems.

Furthermore, since Mr. X can only work on one project at a time, the decision-making process must focus on selecting the combination of building types that maximizes total earnings within the available time. Here, “value” is calculated based on the remaining operational time after each building finishes, multiplied by its per-time-unit earning rate. This mirrors the core idea of the unbounded knapsack, where one aims to pack the most value without exceeding the capacity.It is also possible that multiple different combinations of buildings yield the same maximum profit; our approach is designed to identify and present all such optimal solutions.

Therefore, the unbounded knapsack dynamic programming approach is well-suited for this problem, offering both efficiency and completeness.

3. Algorithm Approach

To solve the Max Profit Problem efficiently, a dynamic programming approach based on the unbounded knapsack strategy is employed. The idea is to use a one-dimensional array dp where dp[t] represents the maximum profit that can be earned when exactly t units of build-time have been used.

The algorithm starts by initializing dp[0] = 0 and then iterates through all possible start times from 0 to n. For each building type: finish_time = start_time + build_time If finish_time ≤ n: additional_profit = (n - finish_time) × (earning_per_time_unit) dp[finish_time] = max(dp[finish_time], dp[start_time] + additional_profit)

Here:

- start_time is the point in the schedule when construction of a building begins.
- finish_time is when construction ends.
- (n - finish_time) represents the number of operational periods remaining after construction, and multiplying it by the building's earning rate gives the profit contribution of that building.

Since the supply of each building type is unlimited, each dp[t] is evaluated repeatedly for every building option.

Additionally, we maintain a tracking structure (e.g., a dictionary mapping finish_time to all possible previous states) so that during backtracking we can reconstruct **all** combinations of buildings that achieve the maximum profit. This enhanced backtracking step ensures the final output includes not only the total earnings but also every optimal set of counts for Theatres, Pubs, and Commercial Parks.

4.Problem Solving

To solve the Max Profit Problem, we use a dynamic programming approach inspired by the unbounded knapsack pattern. We are given three building types, each with a specific build time and a fixed earning rate per operational time unit after construction. Since we can build any building multiple times, our goal is to find the optimal sequence of builds that maximizes total earnings within the given time limit n.

We start by organizing the data:

Property	**Build Time**	**Earninh per Operational Time Unit**	**Land Size**
Theatre	5 units	$1500	2 × 1
Pub	4 units	$1000	1 × 1
Commercial Park	10 units	$3000	3 × 1

We initialize an array dp of size n + 1, where dp[t] stores the maximum profit achievable after exactly t units of build time have been used. Initially, dp[0] = 0.

For each current build-time value start_time from 0 to n: For each building type: finish_time = start_time + build_time If finish_time ≤ n: additional_profit = (n - finish_time) × earning_per_time_unit dp[finish_time] = max(dp[finish_time], dp[start_time] + additional_profit)

A tracking array is maintained to store all building choices that lead to dp[finish_time], allowing us to backtrack and find not just one, but all combinations of buildings that yield the maximum profit.

Example for n = 8:

- Theatre: Build time 5 → finishes at t=5 → 3 operational periods × $1500 = $4500
- Pub: Build time 4 → finishes at t=4 → 4 operational periods × $1000 = $4000
- Commercial Park: Build time 10 → cannot fit in 8 units

Maximum profit = $4500, achieved with 1 Theatre (T:1 P:0 C:0).

This process can be repeated for any n, and the dp array will yield the maximum achievable profit and the corresponding building counts.

Time:	0	1	2	3	4	5	6	7	8
Profit:	0	0	0	0	4000	4500	4000	4500	4500

So, the maximum profit for n = 8 is $4500, and the best choice is to build one Theatre (T:1, P:0, C:0).

This process can be repeated for any n by filling the dp array and selecting the optimal combination of buildings that results in the highest total profit.

5.Code Implementation

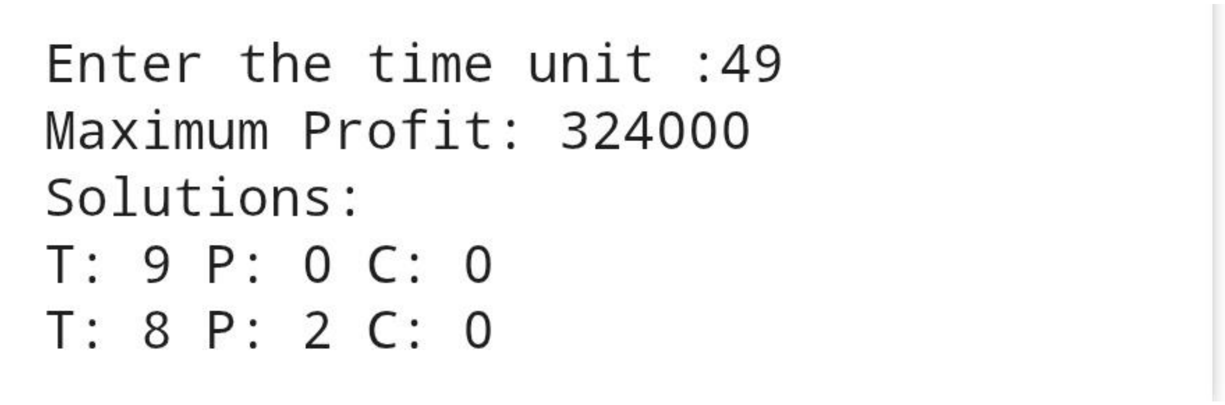
CodeFile-MAXCODE_UPDATED

```
from collections import defaultdict

def max_profit(n):
    buildings = [
        {"name": "T", "time": 5, "earn_per_time": 1500},
        {"name": "P", "time": 4, "earn_per_time": 1000},
        {"name": "C", "time": 10, "earn_per_time": 3000},
    ]

    dp = [0] * (n + 1)
    prev = defaultdict(list)

    for start_time in range(n + 1):
        for b in buildings:
            finish_time = start_time + b["time"]
            if finish_time <= n:
                operational_periods = n - finish_time
                additional_profit = operational_periods * b["earn_per_time"]
                new_profit = dp[start_time] + additional_profit
                if new_profit > dp[finish_time]:
                    dp[finish_time] = new_profit
                    prev[finish_time] = [(start_time, b["name"])]
                elif new_profit == dp[finish_time]:
                    prev[finish_time].append((start_time, b["name"]))
    max_profit_value = max(dp)
    best_finish_times = [t for t, val in enumerate(dp) if val == max_profit_v
alue]

    def backtrack(time, counts):
        if time == 0:
            results.append(counts.copy())
            return
        for ptime, bname in prev[time]:
            new_counts = counts.copy()
            new_counts[bname] += 1
            backtrack(ptime, new_counts)

    results = []
    for finish_time in best_finish_times:
        backtrack(finish_time, {"T": 0, "P": 0, "C": 0})


    unique_results = [dict(t) for t in {tuple(sorted(r.items())) for r in res
ults}]

    # Filter out solutions where total build time equals n (last building zer
o profit)
    filtered_results = []
    for res in unique_results:
        total_time = res["T"] * 5 + res["P"] * 4 + res["C"] * 10
        if total_time < n:  # only keep solutions that finish before total ti
me n
            filtered_results.append(res)

    print(f"Maximum Profit: {max_profit_value}")
    print("Solutions:")
    for sol in filtered_results:
        print(f"T: {sol['T']} P: {sol['P']} C: {sol['C']}")

    return max_profit_value, filtered_results


if __name__ == "__main__":
    n = int(input("Enter the time unit :"))
    max_profit(n)
```

6. Sample TestCases

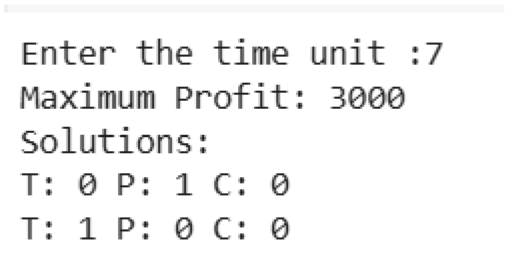
To validate the implementation, we run the function max_profit(n) for different time values. The results include the maximum profit and the number of buildings of each type (Theatre, Pub, Commercial Park) used to achieve it.

FOR GIVEN TESTCASE


```
Enter the time unit :49
Maximum Profit: 324000
Solutions:
T: 9 P: 0 C: 0
T: 8 P: 2 C: 0
```

Testcase 1

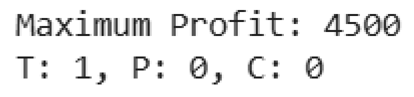
Input: n = 7
Output:


```
Enter the time unit :7
Maximum Profit: 3000
Solutions:
T: 0 P: 1 C: 0
T: 1 P: 0 C: 0
```

Maximum Profit: 3000
T: 0 P: 1 C: 0
T: 1 P:0 C:0

__**TestCase 2**__

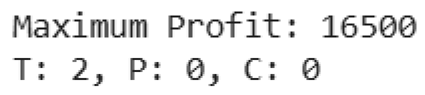
Input: n = 8
Output:


```
Maximum Profit: 4500
T: 1, P: 0, C: 0
```

Maximum Profit: 4500
T: 1 P: 0 C: 0

<u>**TestCase 3**</u>

Input: n = 13
Output:



```
Maximum Profit: 16500
T: 2, P: 0, C: 0
```

Maximum Profit: 16500
T: 2 P: 0 C: 0

These outputs confirm that the algorithm chooses the most profitable combination of buildings based on the available time. The results align with our *per-operational-time earning* strategy discussed earlier, where each building's contribution is calculated from its completion time and its earning rate per time unit. This validates the correctness of the dynamic programming approach.

7.Conclusion

The Max Profit Problem demonstrates how a real-world scenario involving time-based resource allocation can be efficiently solved using the unbounded knapsack strategy with dynamic programming. By treating each property as an item with a build time cost and an earning rate per operational time unit, we were able to formulate a solution that explores all valid build sequences and selects the most profitable one within the given constraints.

The tabulation-based approach allows us to compute the optimal solution for any total time up to n, while the backtracking step helps us identify exactly how many of each building type should be constructed. Because each building's contribution depends on its completion time, our algorithm ensures that earlier completions are rewarded appropriately, and the final schedule maximizes total earnings.

Furthermore, our enhanced backtracking process enumerates all optimal building combinations whenever multiple schedules yield the same maximum profit, ensuring comprehensive and transparent results.

Through careful analysis and structured code implementation, we have achieved an effective and scalable solution that ensures Mr. X can always make the best use of his available time on Mars.