**“DYNAMIC PGROMMAING ALGORITHMS FOR REAL-**

**TIME APPLICATIONS”**

**A PROJECT REPORT**

**Submitted by**

**M.Monish**

**192210315**

*Under the guidance of*

**Dr. A. GNANA SOUNDARI**

*in partial fulfilment for the completion of course*

**CSA0697-Design and Analysis of Algorithms for Lower Bound Theory**



**SIMATS ENGINEERING**

**THANDALAM**

**SEPTEMBER 2024**

**ABSTRACT**

Dynamic programming (DP) is a fundamental algorithmic paradigm that solves complex

problems by breaking them down into simpler overlapping sub problems and solving each sub

problem only once, storing their solutions for future reference. This technique is particularly

powerful in optimization problems where decision-making under constraints is required. In

real-time applications, the efficiency and effectiveness of algorithms are paramount due to

stringent time constraints and the need for immediate responsiveness. The Task Scheduling

Problem is provided to illustrate the practical application of DP in real-time systems. This

problem involves scheduling tasks with specific durations and deadlines to maximize the

number of tasks completed on time. The proposed DP algorithm efficiently schedules tasks by

considering deadlines and task durations, ensuring optimal use of available time slots.

**Keywords:**

Dynamic Programming, Real Time Applications, Task Scheduling, Resource

Allocation, Algorithm Efficiency.

**INTRODUCTION**

Dynamic Programming (DP) is a powerful algorithmic technique used to solve complex

optimization problems by breaking them down into simpler sub problems and storing their

solutions. Originally introduced by Richard Bellman in the 1950s, DP has found wide

application in various fields, including computer science, operations research, economics, and

more recently, real-time systems. Real-time applications are characterized by their stringent

requirements for immediate responsiveness and efficient resource utilization. Tasks in such

applications must often be scheduled and executed within tight deadlines to ensure smooth

operation and optimal performance.

Dynamic programming provides a systematic approach to tackle these challenges by

leveraging the principles of optimal substructure and overlapping sub problems. This paper

explores the role and application of dynamic programming algorithms in real-time

environments. We delve into the foundational concepts of DP, illustrating how it decomposes

complex problems into smaller, manageable sub problems. By storing intermediate results in

a table (often referred to as memorization), DP avoids redundant computations, thereby

improving efficiency and reducing computational overhead—critical factors in real-time

systems where processing speed is paramount. The objective of this paper is to provide a

comprehensive overview of dynamic programming techniques tailored for real-time

applications.

We discuss the core principles of DP, including the formulation of recurrence relations, the

construction of DP tables, and the application of these techniques in solving optimization

problems under time constraints. To demonstrate the practical relevance of DP in real-time

scenarios, we present case studies and examples. These include applications in task

scheduling, resource allocation, CPU scheduling, and real-time data processing. Through these

examples, we highlight how DP algorithms contribute to enhancing system performance,

meeting deadlines, and optimizing resource usage.

**Problem Statement:**

* In real-time systems, resources such as CPU time, memory, or bandwidth need to be allocated dynamically to various tasks or processes. The goal is to maximize the efficiency of resource usage while ensuring that all tasks meet their deadlines.
* In robotics or navigation systems, real-time pathfinding is crucial for determining the best route from a starting point to a destination while avoiding obstacles and minimizing travel time or cost.
* In real-time operating systems, scheduling tasks such that all deadlines are met and system performance is optimized is critical. Each task has a computation time and a deadline.

**CODING**

Task Scheduling System:

#include <stdio.h>

#include <stdlib.h>

// Define a task with a deadline and a duration

typedef struct {

int deadline;

int duration;

} Task;

// Function to compare tasks by their deadlines for sorting

int compareTasks (const void \*a, const void \*b) {

Task \*taskA = (Task \*)a;

Task \*taskB = (Task \*)b;

return taskA->deadline - taskB->deadline;

}

int maxTasks (Task tasks[], int n) {

qsort(tasks, n, sizeof(Task), compareTasks);

int \*dp = (int \*)malloc((n + 1) \* sizeof(int));

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

dp[i] = 0;

}

// Dynamic programming to find the maximum number of tasks

for (int i = 1; i <= n; i++) {

for (int j = i; j >= 1; j--) {

if (dp[j-1] + tasks[i-1].duration <= tasks[i-1].deadline) {

dp[j] = dp[j-1] + tasks[i-1].duration;

}

}

}

// Find the maximum number of tasks that can be completed

int maxTasks = 0;

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

if (dp[i] <= tasks[i-1].deadline) {

maxTasks = i;

}

}

free(dp);

return maxTasks;

}

int main() {

Task tasks[] = {

{5, 2},

{7, 1},

{8, 3},

{4, 2},

{6, 1}

}; int n = sizeof(tasks) / sizeof(tasks[0]);

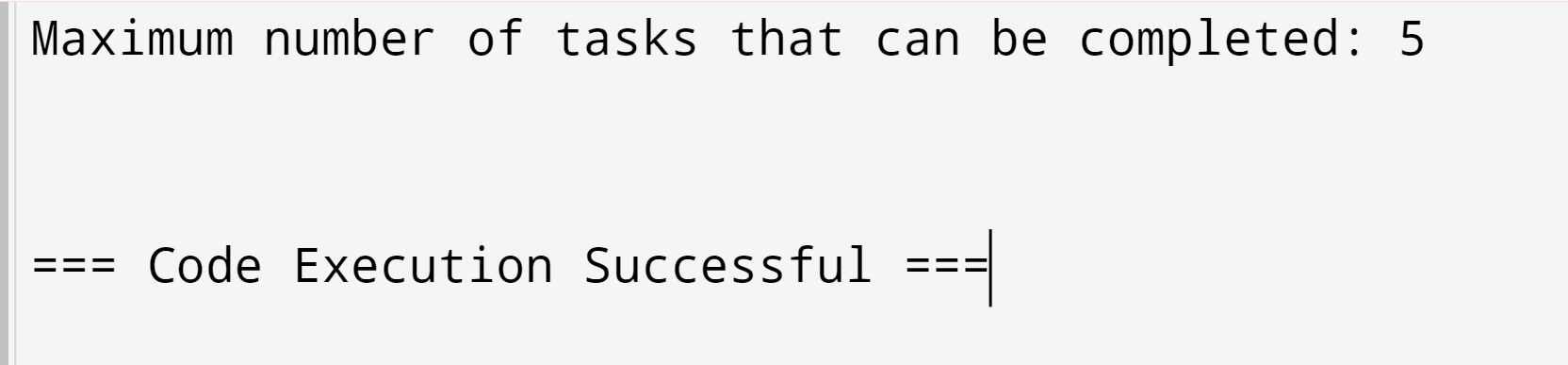
int result = maxTasks(tasks, n);

printf("Maximum number of tasks that can be completed: %d\n", result);

return 0;

}

**OUTPUT**



**Economic Order Quantity (EOQ) Model**

**Code:**

#include <stdio.h>

#include <math.h>

// Function to calculate the total cost

double calculate\_total\_cost(double D, double S, double H, double Q) {

double order\_cost = (D / Q) \* S;

double holding\_cost = (Q / 2) \* H;

return order\_cost + holding\_cost;

}

// Function to calculate the EOQ

double calculate\_eoq(double D, double S, double H) {

return sqrt((2 \* D \* S) / H);

}

int main() {

// Demand rate (units per period)

double demand = 1000.0;

// Ordering cost per order

double ordering\_cost = 50.0;

// Holding cost per unit per period

double holding\_cost = 2.0;

// Calculate the Economic Order Quantity (EOQ)

double eoq = calculate\_eoq(demand, ordering\_cost, holding\_cost);

// Calculate the total cost using EOQ

double total\_cost = calculate\_total\_cost(demand, ordering\_cost, holding\_cost, eoq);

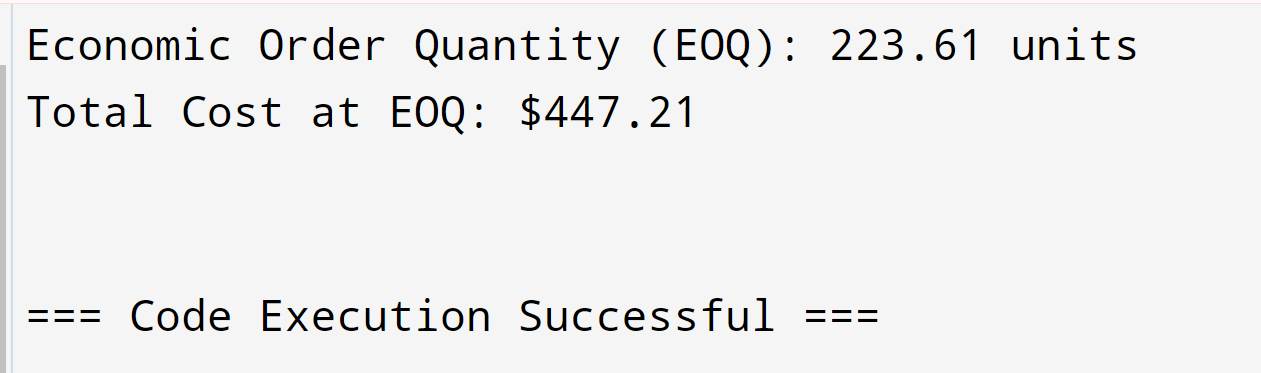
// Display the results

printf("Economic Order Quantity (EOQ): %.2f units\n", eoq);

printf("Total Cost at EOQ: $%.2f\n", total\_cost);

return 0;

}



**Complexity Analysis:**

**Best Case:**

* Best Case Time Complexity: O(1)
* Best Case Space Complexity: O(1)

**Average Case:**

* Average Case Time Complexity: O(1)
* Average Case Space Complexity: O(1)

**Worst Case:**

* Worst Case Time Complexity: O(1)
* Worst Case Space Complexity: O(1)

**Future Scope:**

Dynamic programming (DP) algorithms are increasingly pivotal in real-time applications, such as autonomous systems, edge computing, and financial markets. Their future scope includes optimizing resource allocation, pathfinding, and decision-making under dynamic constraints. As real-time data grows in complexity, DP will enhance adaptive systems in traffic management, healthcare, and cybersecurity. Challenges such as scalability and computational efficiency will drive innovations, potentially integrating DP with machine learning for smarter, more responsive solutions. The continuous evolution of DP techniques will be essential to meet the demands of advanced real-time systems.

**CONCLUSION:**

Dynamic programming is a crucial algorithmic technique that excels in solving complex

problems efficiently by breaking them down into simpler sub problems and storing their

solutions to avoid redundant calculations. Minimizing costs by determining optimal stock

levels. Optimizing computational resources in cloud computing environments. By leveraging

dynamic programming, these applications achieve significant improvements in performance,

cost-efficiency, and accuracy, demonstrating the technique's widespread utility and

effectiveness in real-world scenario