# Data Structure and Algorithm Design

**ME/MSc (Computer) – Pokhara University**


**Prepared by:**

Assoc. Prof. Madan Kadariya (NCIT)

Contact: madan.kadariya@ncit.edu.np

# Chapter 6: Dynamic Programming and Greedy Algorithms(6 hrs)

# Outline

**6. Dynamic Programming and Greedy Algorithms (6 hrs)**

6.1. Engineering applications of dynamic programming: Resource Scheduling

6.2. Greedy algorithms: Huffman coding and its relevance to data compression in embedded systems

6.3. Optimizing dynamic programming techniques for real-time systems

# Dynamic Programming

- **Dynamic programming:** like the **divide-and-conquer** method, dynamic programming solves problems by combining the solutions to subproblems. (Programming refers to the tabular method, not to writing computer code.)
- In contrast to divide and conquer, dynamic programming applies when the subproblems overlap- that is, when subproblems share subsubproblems.
- In this context, a divide-and-conquer algorithm does more work than necessary, repeatedly solving the common subsubproblems.
- A dynamic-programming algorithm solves each subsubproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time it solves each subsubproblem.
- Dynamic Programming solves optimization problems by breaking them into overlapping subproblems and storing solutions to avoid recomputation. It finds an optimal solution (maximum or minimum value) among multiple possible solutions.

# Dynamic Programming

To develop a dynamic-programming algorithm, follow a sequence of four steps:

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution, typically in a bottom-up fashion.
4. Construct an optimal solution from computed information.

- Steps 1-3 form the basis of a dynamic-programming solution to a problem. If we need only the value of an optimal solution, and not the solution itself, then we can omit step 4.
- When we do perform step 4, it often pays to maintain additional information during step 3 so that we can easily construct an optimal solution. 50

# Dynamic Programming

## Key Concepts of Dynamic Programming Algorithm

### 1. Overlapping Subproblems

- Overlapping subproblems in dynamic programming occur when a problem is broken into smaller subproblems that are solved multiple times. Instead of recomputing, dynamic programming solves each subproblem once, stores the result, and reuses it when needed.
- For example, in the Fibonacci sequence, calculating Fib(5) requires Fib(4) and Fib(3), while Fib(4) also requires Fib(3) and Fib(2). Without dynamic programming, these calculations repeat unnecessarily. By storing results like Fib(2) and Fib(3), we avoid redundancy, making the algorithm more efficient.

### 2. Optimal Substructure

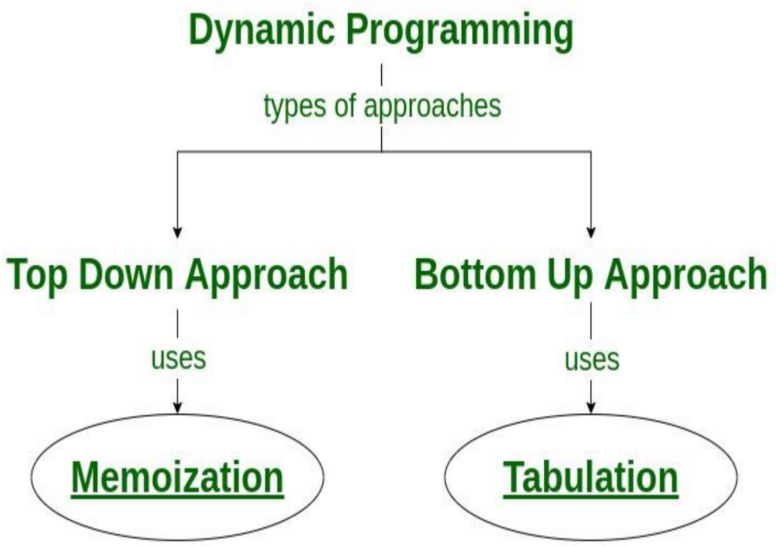
- Optimal substructure means an optimal solution can be constructed from optimal solutions to its subproblems.
- For example, in the shortest path problem, if the shortest path from A to B and B to C is known, then the shortest path from A to C is their combination. Dynamic programming leverages this by solving and combining smaller subproblems to form the final solution.

# Dynamic Programming

## Approaches of Dynamic Programming

### 1. Top-Down Approach (Memoization)

- The top-down approach starts with the main problem and breaks it into smaller subproblems recursively. Each solved subproblem is stored (memoized) in a table or cache to avoid redundant calculations.
- **Example:**
  Computing Fibonacci numbers using recursion while storing previously computed values to improve efficiency.




### 2. Bottom-Up Approach (Tabulation)

- The bottom-up approach solves the smallest subproblems first and uses their results to build the solution iteratively. A table (tabulation) is filled until the final solution is reached, making it more space-efficient and avoiding recursive overhead.
- **Example:**
  Computing Fibonacci numbers iteratively by filling an array from Fib(0) and Fib(1) up to Fib(n).

# Dynamic Programming

## Common Dynamic Programming Example Problems

### 1. Fibonacci Sequence

- **Problem:** Calculate the ***nth*** Fibonacci number where each number is the sum of the two preceding ones, starting from 0 and 1.
- **DP Approach:** Use a simple recurrence relation ***Fib(n) = Fib(n-1) + Fib(n-2)*** with memoization or tabulation to avoid repeated calculations.

### 2. 0/1 Knapsack Problem

- **Problem:** Given weights and values of items, determine the maximum value that can be obtained by selecting items without exceeding a given weight capacity.
- **DP Approach:** Use a 2D table to store the maximum value that can be achieved for each sub-capacity and subset of items, using the recurrence relation that considers whether to include or exclude each item.

### 3. Longest Common Subsequence (LCS)

- **Problem:** Given two sequences, find the length of the longest subsequence that is common to both sequences.
- **DP Approach:** Use a 2D table where ***LCS[i][j]*** represents the length of the LCS of the first ***i*** characters of the first sequence and the first ***j*** characters of the second sequence.

# Dynamic Programming

## Common Dynamic Programming Example Problems

### 4. Coin Change Problem

- **Problem:** Given a set of coin denominations and a total amount, determine the minimum number of coins needed to make the amount.
- **DP Approach:** Use a 1D table where each entry represents the minimum number of coins needed to make that amount, iteratively building up the solution.

### 5. Rod Cutting Problem

- **Problem:** Given a rod of a certain length and prices for different lengths, determine the maximum profit that can be obtained by cutting the rod and selling the pieces.
- **DP Approach:** Use a 1D table to store the maximum profit for each length of the rod, considering all possible cuts.

### 6. Matrix Chain Multiplication

- **Problem:** Given a sequence of matrices, find the most efficient way to multiply them together by determining the optimal order of multiplication.
- **DP Approach:** Use a 2D table to store the minimum number of scalar multiplications needed to multiply the matrices, considering all possible ways to split the sequence.

# Dynamic Programming

## Dynamic Programming Vs Recursion

**Recursion** solves a problem by breaking it into smaller subproblems while dynamic programming optimizes this process by storing the results of subproblems (using memoization or tabulation) to avoid solving the same subproblem multiple times.

## Dynamic Programming Vs Greedy Algorithm

**Dynamic Programming** considers all possibilities and stores results to ensure the best solution where as **Greedy Algorithms** make the best immediate choice without considering future consequences, which works only if the greedy choice property holds.

# Dynamic Programming

## Dynamic Programming in Resource Scheduling

- **Resource Scheduling** involves efficiently allocating limited resources (e.g., CPU time, memory, bandwidth, workers) to tasks while optimizing a given objective, such as minimizing cost or maximizing throughput.
- **Dynamic Programming (DP)** is widely used in such problems due to its ability to handle overlapping subproblems and optimal substructure.

## Common Resource Scheduling Problems Solved Using DP

1. **Job Scheduling with Deadlines**
    1. Given ***n*** jobs with deadlines and profits, the goal is to schedule them to maximize total profit.
    2. DP helps by considering subproblems where only the first ***i*** jobs are scheduled optimally.

# Dynamic Programming

## Dynamic Programming in Resource Scheduling

2. **Weighted Interval Scheduling**
   1. Jobs have start and end times with associated weights (profits).
   2. DP finds the optimal subset of non-overlapping jobs that maximizes the total weight.

3. **Task Scheduling in Cloud Computing**
   1. DP optimizes resource allocation across virtual machines to minimize cost and response time.

4. **Knapsack-Based Resource Allocation**
   1. Given limited resources, DP determines how to allocate them for maximum benefit.

# Dynamic Programming

## Dynamic Programming in Resource Scheduling

### 1. Job Scheduling with Deadlines

- Job Scheduling with Deadlines is a classic optimization problem where the goal is to schedule a set of jobs to maximize profit while ensuring that each job is completed by its deadline.
- This problem can be efficiently solved using **dynamic programming**.

**Problem Statement:**

- For given $n$ jobs, each with a deadline $d_i$ and a profit $p_i$.
- Each job takes **1 unit of time** to complete.
- Only one job can be scheduled at a time.
- If a job is completed before or on its deadline, earn the profit $p_i$; otherwise, earn nothing.
- The goal is to maximize the total profit.

# Dynamic Programming

## Dynamic Programming in Resource Scheduling

### 1. Job Scheduling with Deadlines (Dynamic Programming Approach)

- We can solve this problem using dynamic programming by following these steps:

**1. Sort the Jobs:**

- First, sort the jobs in ***decreasing order of profit***. This ensures that we prioritize jobs with higher profit.

**2. Define the DP Table:**

- Let ***dp[t]*** represent the maximum profit that can be earned by scheduling jobs within the first ***t*** time slots.
- Initialize ***dp[0] = 0*** (no profit if no jobs are scheduled).

**3. Fill the DP Table:**

- For each job, try to schedule it in the latest available time slot before its deadline. Update the DP table accordingly.

# Dynamic Programming

## Dynamic Programming in Resource Scheduling

### 1. Job Scheduling with Deadlines (Dynamic Programming Approach)

**4. Recurrence Relation:**

- For each job $i$ with deadline $d_i$ and profit $p_i$:
  - Iterate from $t = d_i$ down to $1$.
  - If the time slot $t$ is available (i.e., no job has been scheduled in that slot), update the DP table:

$$dp[t] = max(dp[t], dp[t-1] + p_i)$$

**5. Result:**

- The maximum profit will be the maximum value in the $dp$ array.

# Dynamic Programming

## Dynamic Programming in Resource Scheduling

### Job Scheduling with Deadlines (Dynamic Programming Approach)-Example

| Job Id | Deadline | Profit |
|---|---|---|
| 1 | 2 | 100 |
| 2 | 1 | 50 |
| 3 | 2 | 10 |
| 4 | 1 | 20 |

1. Sort the jobs by profit in decreasing order: *(JobId, Deadline, profit)* :

   [(1, 2, 100), (2, 1, 50), (4, 1, 20), (3, 2, 10)]

2. Initialize ***dp*** array of size *max_deadline + 1* (here, *max_deadline = 2*): ***dp = [0, 0, 0]***
3. Process each job:
   - Job 1: Deadline = 2, Profit = 100
     - Assign to time slot 2: ***dp = [0, 0, 100]***

## Dynamic Programming in Resource Scheduling

### Job Scheduling with Deadlines (Dynamic Programming Approach)-Example

2. Process each job:
   - Job 2: Deadline = 1, Profit = 50
     - Assign to time slot 1:
     - ***dp = [0, 50, 100]***
   - Job 4: Deadline = 1, Profit = 20
     - Cannot be scheduled (time slot 1 is occupied).
   - Job 3: Deadline = 2, Profit = 10
     - Cannot be scheduled (time slot 2 is occupied).

1. Maximum profit: ***dp = [0, 50, 100] = 100 + 50 = 150***

**Complexity:**

- **Time Complexity**: $O(n\log n + n \cdot d)$, where $n$ is the number of jobs and $d$ is the maximum deadline.
  - Sorting takes $O(n\log n)$.
  - Filling the DP table takes $O(n \cdot d)$.

# Try it out

Schedule the following tasks using **dynamic programming approach of job scheduling with deadline**. Also draw Gantt chart and find the max profit.

| Job | Deadline ($d_i$) | Profit ($p_i$) |
|---|---|---|
| A | 4 | 100 |
| B | 1 | 50 |
| C | 2 | 40 |
| D | 2 | 20 |
| E | 3 | 30 |

Ans:

dp=[50,40,30,100] = 220

Gantt Chart =

| B | C | E | A |
|---|---|---|---|

## Dynamic Programming in Resource Scheduling

### 2. Weighted Interval Scheduling

**Weighted Interval Scheduling** is another classic optimization problem where the goal is to schedule a subset of non-overlapping intervals (jobs) to maximize the total weight (profit). This problem can be efficiently solved using **dynamic programming**.

**Problem Statement:**

- For given $n$ intervals, each with a start time $s_i$, finish time $f_i$, and a weight (profit) $w_i$.
- Two intervals are **non-overlapping** if one finishes before the other starts.
- The goal is to select a subset of non-overlapping intervals that maximizes the total weight.

### Dynamic Programming Approach:

**1. Sort the Intervals:**

- Sort the intervals by their **finish time** in ascending order. This allows us to process intervals in a way that ensures we consider non-overlapping intervals.

# Dynamic Programming

## Dynamic Programming in Resource Scheduling

### 2. Define the DP Table:

- Let *dp[i]* represent the maximum weight that can be obtained by considering the first *i* intervals.
- Initialize *dp[0] = 0* (no intervals selected).

### 3. Precompute the Last Compatible Interval:

- For each interval *i*, precompute the last interval *p[i]* that is compatible with it (i.e., the latest interval that finishes before $s_i$). This can be done using binary search for efficiency.

### 4. Fill the DP Table:

- For each interval *i*, decide whether to include it or exclude it:
- If included, add its weight to the weight of the last compatible interval *p[i]*.
- If excluded, carry over the weight from the previous interval *dp[i-1].*
- Update *dp[i]* as the maximum of these two choices. [$dp[i]=max(dp[i-1],weight[i]+dp[p(i)])$ ]

### 5. Result:

- The maximum weight will be *dp[n]*, where *n* is the total number of intervals.

# Dynamic Programming

## Dynamic Programming in Resource Scheduling

### 2. Weighted Interval Scheduling

### Significance of $p_i$

- $p(i)$ is the index of the last job that does not overlap with job $i$.
- If $p(i) > 0$, it means job $i$ has a non-overlapping compatible job that we could include in the optimal solution, because including both jobs together contributes to the maximum weight.
- If $p(i)$ is 0 (or no valid previous job exists), it means job $i$ doesn't have any compatible jobs before it. It is a starting point or independent job but it might still be part of the optimal solution depending on its weight.

### Complexity:

- **Time Complexity**: $O(n\log n)$, where $n$ is the number of intervals.
  - Sorting takes $O(n\log n)$.
  - Binary search for each interval takes $O(\log n)$, so for all intervals, it takes $O(n\log n)$.
- **Space Complexity**: $O(n)$, for storing the $p$ array.

# Dynamic Programming

## Dynamic Programming in Resource Scheduling

### 2. Weighted Interval Scheduling

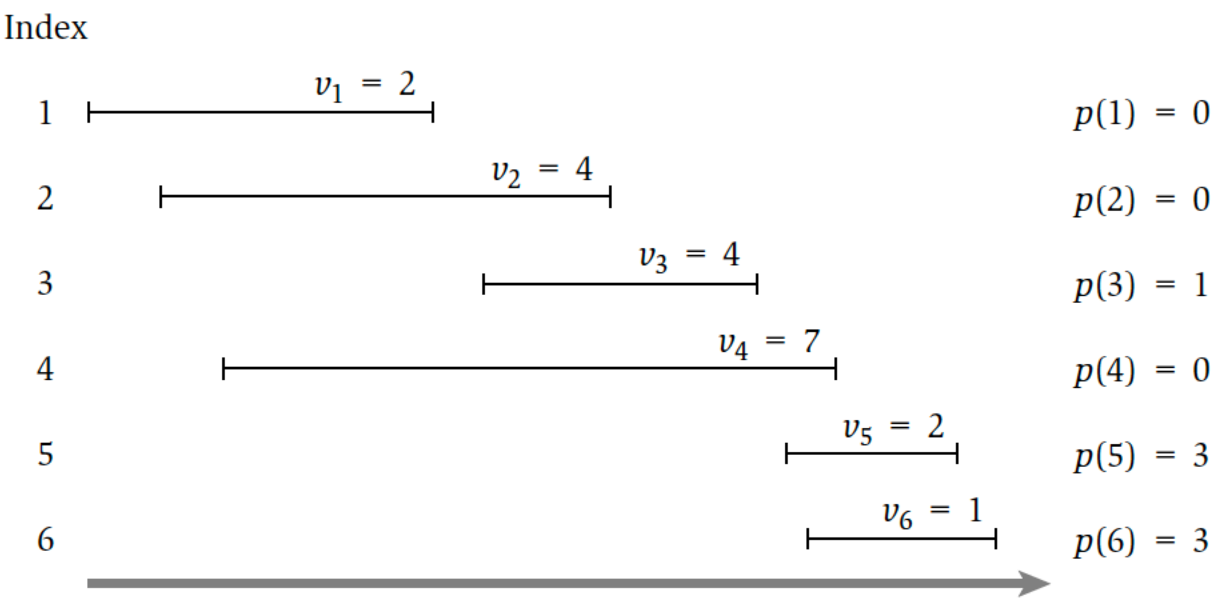



Fig: An instance of weighted interval scheduling with the functions p(i) defined for each interval i.

# Dynamic Programming

## Dynamic Programming in Resource Scheduling

### Weighted Interval Scheduling (Example)

| Jobs | Start time | End time | Weight |
|---|---|---|---|
| A | 1 | 3 | 5 |
| B | 2 | 5 | 6 |
| C | 4 | 6 | 5 |
| D | 6 | 7 | 4 |

Intervals*(start time, end time, weight):*

*[A(1, 3, 5), B(2, 5, 6), C(4, 6, 5), D(6, 7, 4)]*

1. Sort the intervals by finish time: A (1,3,5) → B (2,5,6) → C (4,6,5) → D (6,7,4)
2. Precompute ***p[i]*** (last compatible interval)
   - For interval 1: p[1] = 0 (no compatible interval).
     - [*no task finished before start time i,e 1*]
   - For interval 2: p[2] = 0 (no compatible interval).
     - [*Job A, end time is 3, but Job B starts at 2(Overlap)*]
   - For interval 3: p[3] = 1 (interval 1 is compatible).
     - [Job A, *end time is 3,less than interval 3 start time i.e. 4*]
   - For interval 4: p[4] = 3 (interval 3 is compatible).
     - [Job C, *end time is 6,less than or equal interval 4 start time i.e. 6*]

| Jobs | Start time | End time | Weight | $p_i$ |
|---|---|---|---|---|
| A | 1 | 3 | 5 | 0 |
| B | 2 | 5 | 6 | 0 |
| C | 4 | 6 | 5 | 1 |
| D | 6 | 7 | 4 | 3 |

# Dynamic Programming

## Dynamic Programming in Resource Scheduling

### Weighted Interval Scheduling (Example)

3. Initialize dp array: $dp = [0, 0, 0, 0, 0]$
   - Fill the DP table: We define $dp[i]$ as the **maximum weight possible by selecting from the first** $i$ **jobs**.
     - $dp[i]=max(dp[i-1],weight[i]+dp[p(i)])$
   - **Base Case:** $dp[0] = 0$ (No jobs → 0 weight)

Now, compute $dp[i]$ for each job:

- Job A: $dp[1]=max(dp[0],5+dp[0])=max(0,5+0)=max(0,5)=5$ i,e, $dp[5,0,0,0]$
- Job B: $dp[2]=max(dp[1],6+dp[0])=max(5,6+0)=max(5,6)=6$ i,e, $dp[5,6,0,0]$
- Job C: $dp[3]=max(dp[2],5+dp[1])=max\ (6,5+5)=max(6,10)=10$ i,e, $dp[5,6,10,0]$
- Job D: $dp[4]=max(dp[3],4+dp[3])=max(10,4+10)=max(10,14)=14$ i,e, $dp[5,6,10,14]$

# Dynamic Programming

## Dynamic Programming in Resource Scheduling

### Weighted Interval Scheduling (Example)

| i | Jobs | Start time | End time | Weight | $p_i$ | DP Formula | dp[i] |
|---|---|---|---|---|---|---|---|
| 0 | - | - | - | - | - | Base case | 0 |
| 1 | A | 1 | 3 | 5 | 0 | max(0,5+0) | 5 |
| 2 | B | 2 | 5 | 6 | 0 | max(5,6+0) | 6 |
| 3 | C | 4 | 6 | 5 | 1 | max(6,5+5) | 10 |
| 4 | D | 6 | 7 | 4 | 3 | max(10,4+10) | 14 |

**Backtracking for Optimal Job Selection**

**1.Start at dp[4]**:

1. Since dp[4]>dp[3]; Job D is selected and included in optimal solution.
2. Move to p(4)=3.

**2.At dp[3]>dp[2]=10;** Job C is selected and included in optimal solution.

1. Move to p(3)=1

**3. At dp[1]>dp[0] ;**Job A is selected and included in optimal solution.

1. Move to p(1)=0, and we stop here.

Thus, the **optimal set of jobs are** *{A, C, D}*

**Maximum Total Weight = 14**

# Try it out (Weighted Interval Scheduling)

- From the following set of intervals calculate the **optimal sets of jobs** with **maximum weight** using DP approach of weighted interval scheduling.

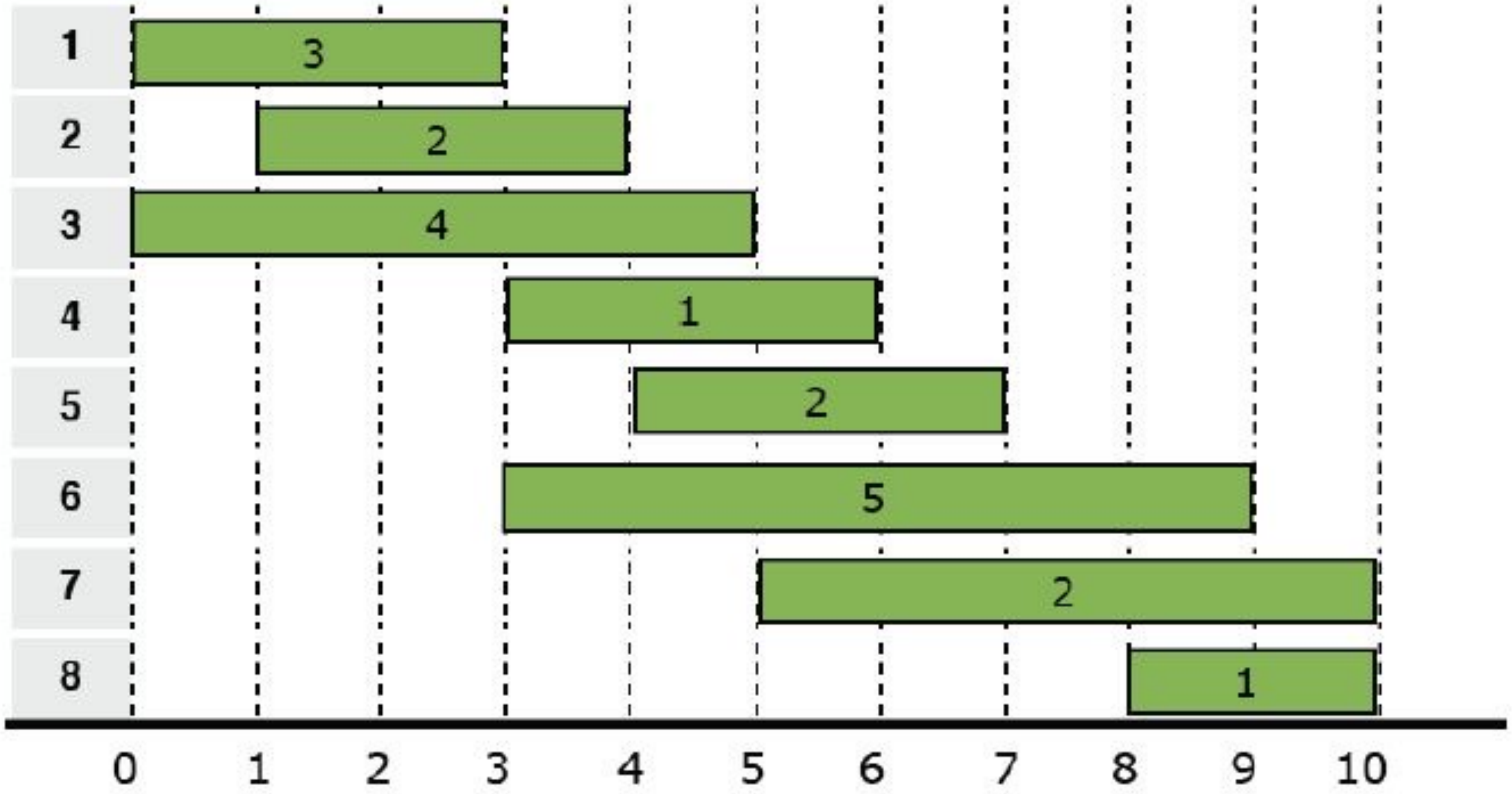
**Steps to Solve the Problem:**

1. Sort intervals by finish time.
2. Find $p(i)$ for each job, where $p(i)$ is the index of the last job that doesn't overlap with job $i$.
3. Use Dynamic Programming (DP) to compute the maximum weight using the recurrence relation:
   $dp[i]=max(dp[i-1],weight(i)+dp[p(i)])$
   where $dp[i-1]$ is the case where job $i$ is not included,
   and $weight(i)+dp[p(i)]$ is the case where job $\boldsymbol{i}$ is included.
4. Backtrack to find the optimal selected jobs.

| Jobs | Start time | Finish Time | Weight |
|---|---|---|---|
| 1 | 0 | 3 | 3 |
| 2 | 1 | 4 | 2 |
| 3 | 0 | 5 | 4 |
| 4 | 3 | 6 | 1 |
| 5 | 4 | 7 | 2 |
| 6 | 3 | 9 | 5 |
| 7 | 5 | 10 | 2 |
| 8 | 8 | 10 | 1 |

# Weighted Interval Scheduling (Example)

Sorted Intervals=[(0,3,3),(1,4,2),(0,5,4),(3,6,1),(4,7,2),(3,9,5),(5,10,2),(8,10,1)] and Gantt chart




**p(i)**
**P(1) = 0**
**p(2) = 0**
**p(3) = 0**
**p(4) = 1**
**p(5) = 2**
**p(6) = 1**
**p(7) = 3**
**p(8) = 5**

# Weighted Interval Scheduling (Example)

- Calculate $dp[i]=max(dp[i-1], weight(i) + dp[p(i)])$

| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| $w_i$ | 3 | 2 | 4 | 1 | 2 | 5 | 2 | 1 |
| $p_i$ | 0 | 0 | 0 | 1 | 2 | 1 | 3 | 5 |
| $dp_i$ | 3 | 3 | 4 | 4 | 5 | 8 | 8 | 8 |

1. **Job 1:** $(0, 3, 3)= dp[1]=max(dp[0],3+dp[0])=max\ (0,3)=3$
2. Job 2: $(1, 4, 2)= dp[2]=max(dp[1],2+dp[0])=max(3,2)=3$
3. Job 3: $(0, 5, 4)=dp[3]=max\ (dp[2],4+dp[0])=max\ (3,4)=4$
4. Job 4: $(3, 6, 1)=dp[4]=max\ (dp[3],1+dp[1])=max(4,1+3)=4$
5. Job 5: $(4, 7, 2)=dp[5]=max\ (dp[4],2+dp[1])=max\ (4,2+3)=5$
6. Job 6: $(3, 9, 5=dp[6]=max(dp[5],5+dp[1])=max(5,5+3)=8$
7. Job 7: $(5, 10, 2)=dp[7]=max\ (dp[6],2+dp[3])=max\ (8,2+4)=8$
8. Job 8: $(8, 10, 1)=dp[8]=max\ (dp[7],1+dp[7])=max\ (8,1+8)=8$

# Weighted Interval Scheduling (Example)

## Backtracking to Find the Optimal Job Set

- Backtrack to find the **optimal job set**.
- Start with *dp[8] = 8*. Since *dp[8]=dp[7]*, Job 8 is not included.
  - Move to *dp[7] = 8*. Since *dp[7]=dp[6]*, Job 7 is not included.
  - Move to *dp[6] = 8*. Since *dp[6]>dp[5]*, *Job 6 is included in optimal job*.
  - Move to *p(6) = 1*.
- At *dp[1] = 3*, since *dp[1]>dp[0]*, *Job 1 is included in optimal job*.
  - Move to *p(1) = 0*.
- Now, the optimal selected jobs are *Job 1* and *Job 6*.

**Final Answer:**

- **Optimal Job Set**: *Jobs 1 and 6.*
- **Maximum Total Weight**: *8.*

# Dynamic Programming

## Dynamic Programming in Resource Scheduling

### 3. Task Scheduling in Cloud Computing

- Task scheduling in **cloud computing** is a crucial problem that involves allocating tasks to virtual machines (VMs) optimally to achieve objectives such as:
  - Minimizing execution time
  - Minimizing cost
  - Maximizing resource utilization
  - Balancing workload across VMs

**Problem Statement:**

- A set of **tasks** $T=\{T_1,T_2,\ldots,T_n\}$, where each task $T_i$ has a **processing time** $p_i$.
- A set of **resources** $R=\{R_1,R_2,\ldots,R_m\}$, where each resource $R_j$ has a **capacity** $c_j$.
- The goal is to **assign tasks to resources** such that:
  - The **total processing time** on each resource does not exceed its capacity.
  - The **makespan** (maximum completion time across all resources) is minimized.

# Dynamic Programming

## 3. Task Scheduling in Cloud Computing (Dynamic Programming Approach)

**Steps:**

**1. Define the DP State:**

- Let $dp[i][j]$ represent the **minimum makespan** achievable when:
  - The first $i$ tasks are assigned.
  - The $j$-th resource has a **current load** of $l_j$.

**2. Initialize the DP Table:**

  - $dp[0][j]=0$ for all $j$ (no tasks assigned, so all resources have zero load).

**3. Recurrence Relation:**

- For each task $T_i$ and each resource $R_j$, update the DP table as follows:
  - $dp[i][j]=min(dp[i-1][j],max(dp[i-1][j],l_j+p_i))$ where:
  - $l_j$ is the current load on resource $R_j$.
  - $p_i$ is the processing time of task $T_i$.

**4. Final Result:**

- The **minimum makespan** is the minimum value in the last row of the DP table.

# Dynamic Programming

## 3. Task Scheduling in Cloud Computing (Dynamic Programming Approach)

**Example:**

**Input:**

- Tasks: $T=\{T_1,T_2,T_3\}$ with processing times $p=[2,3,4]$.
- Resources: $R=\{R_1,R_2\}$ with capacities $c=[5,5]$.

**Step 1: Initialize DP Table**

- $dp[0][0]=0$ (no tasks assigned to $R_1$).
- $dp[0][1]=0$ (no tasks assigned to $R_2$).

```
dp = [
  [0, 0],  # Task 0
  [0, 0],  # Task 1
  [0, 0],  # Task 2
  [0, 0]   # Task 3
]
```

**Step 2: Assign Tasks**

**Task 1 ($T_1$, $p_1=2$):**

- Assign $T_1$ to $R_1$:
  - $dp[1][0]=max(dp[0][0],0+2)=2$
- Assign $T_1$ to $R_2$:
  - $dp[1][1]=max(dp[0][1],0+2)=2$

```
dp = [
  [0, 0],  # Task 0
  [2, 2],  # Task 1
  [0, 0],  # Task 2
  [0, 0]   # Task 3
]
```

# Dynamic Programming

## 3. Task Scheduling in Cloud Computing (Dynamic Programming Approach)

**Task 2 ($T_2$, $p_2=3$):**

Assign $T_2$ to $R_1$:

$$dp[2][0]=max(dp[1][0],2+3)=5$$

Assign $T_2$ to $R_2$:

$$dp[2][1]=max(dp[1][1],2+3)=5$$

```
dp = [
  [0, 0],  # Task 0
  [2, 2],  # Task 1
  [5, 5],  # Task 2
  [0, 0]   # Task 3
]
```

**Task 3 ($T_3$, $p_3=4$):**

Assign $T_3$ to $R_1$:

$$dp[3][0]=max(dp[2][0],5+4)=9$$

Assign $T_3$ to $R_2$:

$$dp[3][1]=max(dp[2][1],5+4)=9$$

```
dp = [
  [0, 0],  # Task 0
  [2, 2],  # Task 1
  [5, 5],  # Task 2
  [9, 9]   # Task 3
]
```

# Dynamic Programming

## 3. Task Scheduling in Cloud Computing (Dynamic Programming Approach)

### Step 3: Final Result

- The **minimum makespan** is the minimum value in the last row of the DP table:

$min(dp[3][0],dp[3][1])=min(9,9)=9$

### Optimal Scheduling:

- Assign $T_1$ and $T_2$ to $R_1$:
  - $R_1{:}T_1(2)+T_2(3)=5$
- Assign $T_3$ to $R_2$:
  - $R_2{:}T_3(4)=4$
- **Makespan**: $max(5,4)=5$

```
Final dp = [
  [0, 0],  # Task 0
  [2, 2],  # Task 1
  [5, 5],  # Task 2
  [5, 9]   # Task 3
]
```

**Final Answer:**

- **Minimum Makespan**: $5$
- **Optimal Scheduling**:
  - $R_1{:}T_1(2)+T_2(3)=5$
  - $R_2{:}T_3(4)=4$

1. The DP approach considers all possible ways to assign tasks to resources.
2. The recurrence relation ensures that the makespan is minimized.
3. The final result is the minimum value in the last row of the DP table.

# Summary of Allocation

| Step | Task Assigned | Assignment Details | State (i,c1,c2) | Makespan |
|---|---|---|---|---|
| 1 | None | Initial state (no tasks assigned). | (0,5,5) | 0 |
| 2 | T1 | Assign T1 (p=2) to R1. Remaining: R1=3, R2=5. | (1,3,5) | 2 |
| 3 | T1 | Assign T1 (p=2) to R2. Remaining: R1=5, R2=3. | (1,5,3) | 2 |
| 4 | T2 | Assign T2 (p=3) to R1 from state (1,3,5). | (2,0,5) | 5 |
| 5 | T2 | Assign T2 (p=3) to R2 from state (1,3,5). | (2,3,2) | 3 |
| 6 | T2 | Assign T2 (p=3) to R1 from state (1,5,3). | (2,2,3) | 3 |
| 7 | T2 | Assign T2 (p=3) to R2 from state (1,5,3). | (2,5,0) | 5 |
| 8 | T3 | Assign T3(p=4) to R2 from state (2,0,5). | (3,0,1) | 5 |
| 9 | T3 | Assign T3 (p=4) to R1 from state (2,5,0). | (3,1,0) | 5 |

*State (i,c1,c2) = (task assigned, Remaining Capacity of R1, Remaining Capacity of R2*

| Resource | Assigned Tasks | Total Processing Time |
|---|---|---|
| R1 | T1 (p=2), T2 (p=3) | 5 |
| R2 | T3 (p=4) | 4 |

**Minimum Makespan**: 5

# Dynamic Programming

## Dynamic Programming in Resource Scheduling

### 4. Knapsack-Based Resource Allocation

- Knapsack-Based Resource Allocation is a **DP** approach used to allocate **tasks, jobs, or workloads** to **computing resources** while maximizing utilization and efficiency.
- It is modeled after the **0/1 Knapsack Problem**, where:
  - **Tasks (Jobs)** are like **items** with a given processing time (weight) and profit (value).
  - **Resources (Computing Nodes, Virtual Machines, or Servers)** are like **knapsacks** with a fixed capacity.
  - **Goal**: Assign tasks to resources while ***maximizing the total benefit*** without exceeding resource limits.

# Dynamic Programming

## Dynamic Programming in Resource Scheduling

### 4. Knapsack-Based Resource Allocation

**Step 1: Define the Problem Inputs**

Given:

- **Tasks**: $T=\{T_1,T_2,\ldots,T_n\}$; **Processing time** $p[i]$ for each task $T_i$; **Benefit (value)** $v[i]$ for each $T_i$
- **Resources**: $R=\{R_1,R_2,\ldots,R_m\}$; **Resource capacity** $c[j]$ for each resource $R_j$

**Step 2: Define the DP State**

- Let $dp[i][j]$ represent the maximum benefit achieved by scheduling the first $i$ tasks within resource capacity $j$.

**Step 3: Establish Recurrence Relation**

- For each task $i$, we have two choices:
  - **Exclude the task** $i$:
    - $dp[i][j]=dp[i-1][j]$ (Do not take the task, keep the same maximum benefit as before.)
  - **Include the task** $i$ (if $p[i]\leq j$):
    - $dp[i][j]=max(dp[i-1][j],dp[i-1][j-w_i]+v_i)$ (Take the task and add its value while reducing available capacity.)

## Dynamic Programming in Resource Scheduling

### 4. Knapsack-Based Resource Allocation

**Step 4: Initialize DP Table**

- Create a $(n+1) \times (c+1)$ DP table initialized with $0$.
- **Base Case**: If no tasks are available $(i = 0)$, the benefit is $0$ for all capacities.

**Step 5: Fill the DP Table**

- Use a nested loop:
  1. Iterate over tasks $i=1$ to $n$
  2. Iterate over capacities $j=1$ to $c$
  3. Apply the recurrence relation.

**Step 6: Extract Optimal Benefit**

- The maximum benefit is stored in $dp[n][c]$ (bottom-right of the table).

# Dynamic Programming

## Dynamic Programming in Resource Scheduling

### 4. Knapsack-Based Resource Allocation

**Step 7: Backtrack to Find Selected Tasks**

- To determine which tasks were included:
  1. Start at *dp[n][c]*
  2. If $dp[i][j] \neq dp[i-1][j]$, task $i$ was selected
  3. Reduce capacity: $j = j - p[i]$
  4. Continue until $j=0$

**Step 8: Schedule Tasks**

- Assign selected tasks to available resources.
- Arrange tasks based on start times and durations.

# Dynamic Programming

## Dynamic Programming in Resource Scheduling

### 4. Knapsack-Based Resource Allocation (Knapsack 0/1) – Example)

**Q**: Tasks: $T=\{T_1,T_2,\ldots,T_n\}$; Weights : $w=[2,3,4]$ ; Values (Benefits): $v=[10,20,30]$ Resource Capacity: $c=5$

**Step 1:** Compute $dp[1][j]$ (Only Task $T_1$)

- $T_1$ has **weight** = 2, **value** = 10.

| Capacity j | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| dp[0][j] | 0 | 0 | 0 | 0 | 0 | 0 |
| dp[1][j] | 0 | 0 | 10 | 10 | 10 | 10 |

**Step 2: Compute dp[2][j] (Using T1 & T2)**

- $T_2$ has **weight** = 3, **value** = 20.

| Capacity j | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| dp[0][j] | 0 | 0 | 0 | 0 | 0 | 0 |
| dp[1][j] | 0 | 0 | 10 | 10 | 10 | 10 |
| dp[2][j] | 0 | 0 | 10 | 20 | 20 | 30 |

At $j=5$, choosing $T_1+T_2$ gives value = 30.

# Dynamic Programming

## Dynamic Programming in Resource Scheduling

### 4. Knapsack-Based Resource Allocation (Knapsack 0/1) – Example)

**Step 3**: Compute *dp[3][j]* (Using $T_1, T_2$ & $T_3$)

- $T_3$ has **weight** = 4, **value** = 30.

| Capacity j | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| dp[0][j] | 0 | 0 | 0 | 0 | 0 | 0 |
| dp[1][j] | 0 | 0 | 10 | 10 | 10 | 10 |
| dp[2][j] | 0 | 0 | 10 | 20 | 20 | 30 |
| dp[3][j] | 0 | 0 | 10 | 20 | 30 | 30 |

At $j=5$, choosing $T_3$ alone gives value = 30.

**Final Answer**

- Maximum Total Value = *30*
- Optimal Task Selection: *{T3}*

At $j=5$:

- **Option 1:** Choose $T_1+T_2$ → Total value = 30
- **Option 2:** Choose *T3* alone → Total value = *30*
- **Optimal choice =** *T3* **(less weight, same value).**

# Try it out (0/1 knapsack)

**Q.** Given Tasks $=\{T_1, T_2, T_3, T_4\}$ ; Weights : $w=[3,4,6,5]$ ; Benefits: $v=[2,3,1,4]$

Resource Capacity: $c=8$ *and* $n=4$

| | $p_i$ | $w_i$ | i/w | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| $T_1$ | 2 | 3 | 1 | 0 | 0 | 0 | 2 | 2 | 2 | 2 | 2 | 2 |
| $T_2$ | 3 | 4 | 2 | 0 | 0 | 0 | 2 | 3 | 3 | 3 | 5 | 5 |
| $T_4$ | 4 | 5 | 3 | 0 | 0 | 0 | 2 | 3 | 4 | 4 | 5 | 6 |
| $T_3$ | 1 | 6 | 4 | 0 | 0 | 0 | 2 | 3 | 4 | 4 | 5 | 6 |

**Final Answer**

- Maximum Total benefit Value = *6*
- Optimal Task Selection: *{T1, T4}*

$dp[i][j]=max(dp[i-1][j], dp[i-1][j-w_i]+v_i)$

dp[1][3]=max(dp[0][3],dp[0][3-3]+2)=max(0,2)=2

dp[2][4]=max(dp[1][4],dp[1][4-4]+3)=max(2,0+3)=3

dp[2][7]=max(dp[1][7],dp[1][7-4]+3)=max(2,dp[1][3]+3)=Max(2,2+3)=5

dp[3][5]=max(dp[2][5],dp[2][5-5]+4)=max(3,dp[2][0]+4)=Max(3,0+4)=4

dp[3][8]=max(dp[2][8],dp[2][8-5]+4)=max(5,dp[2][3]+4)=Max(5,2+4)=6

# Try it out (0/1 knapsack)

**Q.** Given Tasks $=\{T_1, T_2, T_3, T_4\}$ ; Weights : $w=[2,3,4,5]$ ; Benefits: $v=[1,2,5,6]$

Resource Capacity: $c=8$ *and* $n=4$

| | $p_i$ | $w_i$ | i/w | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| $T_1$ | 1 | 2 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| $T_2$ | 2 | 3 | 2 | 0 | 0 | 1 | 2 | 2 | 3 | 3 | 3 | 3 |
| $T_4$ | 5 | 4 | 3 | 0 | 0 | 1 | 2 | 5 | 5 | 6 | 7 | 7 |
| $T_3$ | 6 | 5 | 4 | 0 | 0 | 1 | 2 | 5 | 6 | 6 | 7 | 8 |

**Final Answer**

- Maximum Total benefit Value = *8*
- Optimal Task Selection: $\{T2, T4\}$

$dp[i][j]=max(dp[i-1][j], dp[i-1][j-w_i]+v_i)$

dp[1][2]=max(dp[0][2],dp[0][2-2]+1)=max(0,0+1)=1

dp[2][3]=max(dp[1][3],dp[1][3-3]+2)=max(1,0+2)=2

dp[2][5]=max(dp[1][5],dp[1][5-3]+2)=max(1,dp[1][2]+2)=Max(1,1+2)=3

dp[3][5]=max(dp[2][5],dp[2][5-4]+5)=max(3,dp[2][1]+5)=Max(3,0+5)=5

dp[3][6]=max(dp[2][6],dp[2][6-4]+5)=max(3,dp[2][2]+5)=Max(3,1+5)=6

dp[3][7]=max(dp[2][7],dp[2][7-4]+5)=max(3,dp[2][3]+5)=Max(3,2+5)=7

dp[4][5]=max(dp[3][5],dp[3][5-5]+6)=max(5,dp[3][0]+6)=Max(5,0+6)=6

dp[4][8]=max(dp[3][8],dp[3][8-5]+6)=max(7,dp[3][3]+6)=Max(7,2+6)=8

# Outline

## 6. Dynamic Programming and Greedy Algorithms (6 hrs)

6.1. Engineering applications of dynamic programming: Resource Scheduling

6.2. Greedy algorithms: Huffman coding and its relevance to data compression in embedded systems

6.3. Optimizing dynamic programming techniques for real-time systems

# Huffman coding and its relevance to data compression in embedded systems

# Huffman coding and its relevance to data compression in embedded systems

- Huffman coding is a widely used lossless data compression algorithm that efficiently minimizes average code length by assigning variable-length codes to input characters based on their frequency, making it particularly relevant for embedded systems.

## How Huffman Coding Works

1. **Frequency Analysis**: Count the frequency of each symbol in the input data.
2. **Build a Huffman Tree**:
   - Create a min-heap (priority queue) of nodes, where each node represents a symbol and its frequency.
   - Repeatedly combine the two nodes with the lowest frequencies into a new node until a single tree is formed.
3. **Generate Codes**:
   - Traverse the Huffman tree to assign binary codes to each symbol. Left branches represent 0, and right branches represent 1.
4. **Encode Data**:
   - Replace each symbol in the input data with its corresponding Huffman code.
5. **Decode Data**:
   - Use the Huffman tree to reconstruct the original data from the compressed binary stream.

# Huffman coding and its relevance to data compression in embedded systems

## Relevance to Embedded Systems

Embedded systems often have limited resources, such as memory, processing power, and energy.

1. *Low Computational Overhead:* Huffman coding is simple to implement with efficient encoding and decoding processes with a time complexity of $O(n\log n)$ for building the Huffman tree and $O(n)$ for encoding/decoding.
2. *Memory Efficiency:* The Huffman tree can be compactly represented, and the frequency table can be stored efficiently. Huffman coding can be implemented with minimal memory usage.
3. *Adaptability:* Huffman coding can be adapted to specific data patterns by using a precomputed static Huffman tree for known distributions, eliminating runtime frequency analysis, or employing dynamic Huffman coding for real-time data streams with unknown frequency distributions.
4. *Lossless Compression:* Huffman coding, being inherently lossless, is ideal for embedded systems that require data integrity, making it suitable for applications such as firmware updates, sensor data storage, and communication protocols.
5. *Energy Efficiency:* Huffman coding reduces data transmission and storage requirements, lowering energy consumption in wireless communication and flash memory operations, which is crucial for battery-powered embedded devices.

# Huffman coding and its relevance to data compression in embedded systems

## Applications in Embedded Systems

1. ***Sensor Data Compression:***
   - In IoT devices, sensor data (e.g., temperature, humidity) often contains repetitive patterns that can be efficiently compressed using Huffman coding.
2. ***Firmware Updates:***
   - Compressing firmware updates reduces the amount of data transmitted over the network, saving bandwidth and energy.
3. ***Communication Protocols:***
   - Huffman coding can be used to compress data packets in communication protocols like Zigbee, Bluetooth, or LoRa, improving transmission efficiency.
4. ***Storage Optimization:***
   - In systems with limited flash memory, Huffman coding can compress log files, configuration data, or other stored information.

# Huffman coding and its relevance to data compression in embedded systems

## Optimizations for Embedded Systems

1. ***Static Huffman Trees:***
   - Use precomputed Huffman trees for known data distributions to avoid runtime tree construction.
2. ***Canonical Huffman Coding:***
   - Represent Huffman codes in a compact form, reducing memory usage and simplifying decoding.
3. ***Hardware Acceleration:***
   - Implement Huffman coding in hardware (e.g., using FPGAs or custom ASICs) for faster performance.
4. ***Hybrid Approaches:***
   - Combine Huffman coding with other compression techniques (e.g., Run-Length Encoding or LZ77) for better compression ratios.

# Huffman coding and its relevance to data compression in embedded systems

*Example: Huffman Coding in an IoT-Based Smart Sensor Network*

**Scenario** : A smart agriculture system consists of battery-powered wireless temperature and humidity sensors that transmit data to a central server.

**Challenges**

- **Limited battery life** (radio transmissions consume energy)
- **Low bandwidth availability** (remote areas with weak connectivity)
- **Need for real-time transmission** (climate monitoring)

**Solution:** Implementing Huffman Coding

1. **Data Analysis**:
    1. Temperature values (in °C) range from **15 to 45**.
    2. Humidity values (in %) range from **30 to 90**.
    3. Frequent values were identified and assigned **short Huffman codes**.

# Huffman coding and its relevance to data compression in embedded systems

*Example: Huffman Coding in an IoT-Based Smart Sensor Network*

## 2. Compression Results:

1. **Uncompressed Data**: 16 bits per reading (8 bits for temperature + 8 bits for humidity).
2. **Huffman Compressed Data**: Average **10 bits per reading** (37.5% reduction).
3. **Energy Savings**: Less transmission power required, leading to **20% longer battery life**.

## Outcome

- The Huffman-based compression algorithm allowed faster, more efficient data transmission, extending battery life and improving data collection in remote locations.

## Limitations and Alternatives

- *Requires frequency analysis*, which adds preprocessing overhead.
- *Arithmetic coding and dictionary-based methods* **(e.g., LZ77, LZW)** may provide better compression for certain data types.

# Outline

## 6. Dynamic Programming and Greedy Algorithms (6 hrs)

6.1. Engineering applications of dynamic programming: Resource Scheduling

6.2. Greedy algorithms: Huffman coding and its relevance to data compression in embedded systems

6.3. Optimizing dynamic programming techniques for real-time systems

# Dynamic Programming

## Optimizing dynamic programming techniques for real-time systems

Optimizing DP techniques for RTS requires careful consideration of time constraints, memory efficiency, and computational overhead.

Key strategies to make DP more suitable for real-time applications

### 1. Time Complexity Reduction

- **Use Iterative DP (Bottom-Up) Instead of Recursion**
  Recursive DP (Top-Down with Memoization) introduces function call overhead, which may be problematic for real-time constraints. An iterative approach avoids recursion stack overhead and ensures predictable execution time.
- **State Space Reduction**
  Identify redundant states and reduce the state space where possible. This can be achieved by recognizing overlapping subproblems and eliminating unnecessary computations.
- **Approximate Solutions**
  If an exact solution is not required, approximate DP methods like heuristic-based pruning or greedy approximations can reduce computation time.

# Dynamic Programming

## Optimizing dynamic programming techniques for real-time systems

### 2. Memory Optimization

- **Space-Efficient DP (Rolling Array Technique)**
  Instead of storing full DP tables, use rolling arrays to keep only necessary states. For example, in a DP table of size $O(n)$, reduce space complexity from $O(n^2)$ to $O(n)$ by keeping only the last two rows or columns.
- **In-Place Computation**
  If the DP table can be modified in-place without affecting future computations, avoid extra memory allocations.
- **Bitmasking for State Representation**
  In problems with limited constraints (e.g., knapsack or subset problems), bitwise operations can optimize space usage significantly.

# Dynamic Programming

## Optimizing dynamic programming techniques for real-time systems

### 3. Parallel and Hardware Acceleration

- **Parallelization**

  Some DP problems can be split into independent subproblems that can run concurrently using multi-threading or parallel computing.

- **GPU Acceleration**

  Matrix-based DP problems (e.g., Floyd-Warshall algorithm for shortest paths) can be accelerated using GPUs with CUDA/OpenCL.

- **Custom Hardware (FPGA/ASIC)**

  Critical real-time applications may use specialized hardware to accelerate DP computations.

# Dynamic Programming

## Optimizing dynamic programming techniques for real-time systems

### 4. Real-Time Scheduling Awareness

- **Deadline-Aware Computation**
  Break computations into smaller time slices to ensure tasks meet real-time deadlines.
- **Preemptive Computation**
  If a DP computation is time-sensitive, implement mechanisms to checkpoint intermediate results and resume efficiently.

### 5. Hybrid Approaches

- **Combining DP with Greedy or Divide-and-Conquer**
  Some problems can be solved more efficiently by combining DP with greedy heuristics or divide-and-conquer strategies to reduce computation complexity.
- **Adaptive DP Techniques**
  Modify the DP approach dynamically based on the input constraints to achieve real-time efficiency.

# Dynamic Programming

## Optimizing dynamic programming techniques for real-time systems

*Example: Optimizing the Knapsack Problem for Real-Time Systems*

1. **State Space Reduction**: Use a 1D array instead of a 2D array for the DP table, as the current state only depends on the previous state.
2. **Early Termination**: Stop computation if the remaining capacity is zero or if the maximum possible value is achieved.
3. **Approximation**: Use a greedy heuristic to pre-fill the knapsack with high-value items, then use DP to refine the solution.

# References


1. **Dynamic Programming**:
   - Cormen, T. H.; Leiserson, C. E.; Rivest, R. L. & Stein, C. (2001), *Introduction to Algorithms* , The MIT Press.
   - https://www.wscubetech.com/resources/dsa/dynamic-programming
2. Various resources Like books, Lecture slides from different universities, Web Links, AI tools etc.

***** End of Chapter 6 *****