



Lecture 8

Dynamic Programming I: Introduction, Memoization, Rod Cutting, Knapsack

CS 161 Design and Analysis of Algorithms

Ioannis Panageas

Dynamic Programming

Technique for solving optimization problems.

Solve problem by solving **sub**-problems and combine:

This is called **Optimal substructure** property.

Dynamic Programming

Technique for solving optimization problems.

Solve problem by solving **sub**-problems and combine:

This is called **Optimal substructure** property.

- **Similar** to divide-and-conquer: **recursion** (for solving sub-problems)
- Sub-problems **overlap**: solve them only **once**!

DP = recursion + re-use (**Memoization**)

Dynamic Programming

Example: Given a positive integer numbers n , compute Fibonacci F_n . Definition: $F_1 = F_2 = 1$ and $F_n = F_{n-1} + F_{n-2}$.

Recursion (slow):

Fib(n)

If $n \leq 2$ **then return** 1

return Fib($n - 1$) + Fib($n - 2$)

Dynamic Programming

Example: Given a positive integer numbers n , compute Fibonacci F_n . Definition: $F_1 = F_2 = 1$ and $F_n = F_{n-1} + F_{n-2}$.

Recursion (slow):

Fib(n)

If $n \leq 2$ **then return** 1

return Fib($n - 1$) + Fib($n - 2$)

Why is it slow?

Dynamic Programming

Example: Given a positive integer numbers n , compute Fibonacci F_n . Definition: $F_1 = F_2 = 1$ and $F_n = F_{n-1} + F_{n-2}$.

Recursion (slow):

Fib(n)

If $n \leq 2$ **then return** 1

return Fib($n - 1$) + Fib($n - 2$)

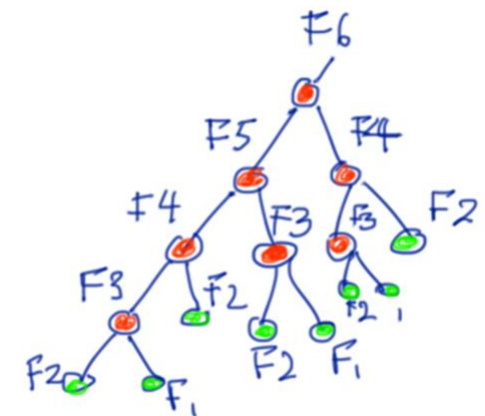
Red nodes: Recursive calls.

Green nodes: Bases cases.

$F(5)$ is computed once, $F(4)$ **twice**,

$F(3)$ **three times**, $F(2)$ **five times**, $F(1)$ **three times**

Why is it slow? $F(6)$



no memoization

Dynamic Programming

Example: Given a positive integer numbers n , compute Fibonacci F_n . Definition: $F_1 = F_2 = 1$ and $F_n = F_{n-1} + F_{n-2}$.

Recursion (slow):

Fib(n)

If $n \leq 2$ **then return** 1

return Fib($n - 1$) + Fib($n - 2$)

Exponential time

Red nodes: Recursive calls.

Green nodes: Bases cases.

F(5) is computed once, F(4) **twice**,

F(3) **three times**, F(2) **five times**, F(1) **three times**

Running time
 $T(n) = T(n - 1) + T(n - 2)$
which is $\Omega(2^{n/2})$

Dynamic Programming

Example: Given a positive integer numbers n , compute Fibonacci F_n . Definition: $F_1 = F_2 = 1$ and $F_n = F_{n-1} + F_{n-2}$.

Memoization (fast):

Array mem[]

Fib(n)

If mem[n] non-empty **then**

return mem[n]

If $n \leq 2$ **then** mem[n] = 1

 mem[n] = Fib($n - 1$) + Fib($n - 2$)

return mem[n]

Dynamic Programming

Example: Given a positive integer numbers n , compute Fibonacci F_n . Definition: $F_1 = F_2 = 1$ and $F_n = F_{n-1} + F_{n-2}$.

Linear time: Let's see $F(6)$

Memoization (fast):

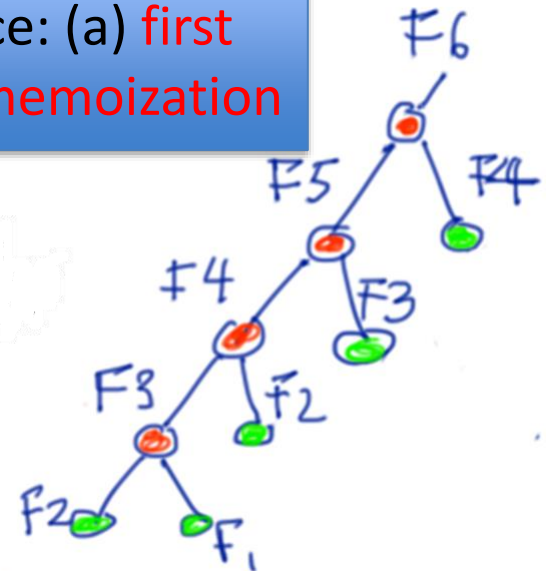
Running time: $\Theta(n)$

Fib(n) will be invoked twice: (a) **first recursion** and (b) **second memoization**

Array mem[]
Fib(n)

If mem[n] non-empty **then**
 return mem[n]

If $n \leq 2$ **then** mem[n] = 1
 mem[n] = Fib($n - 1$) + Fib($n - 2$)
return mem[n]



Dynamic Programming

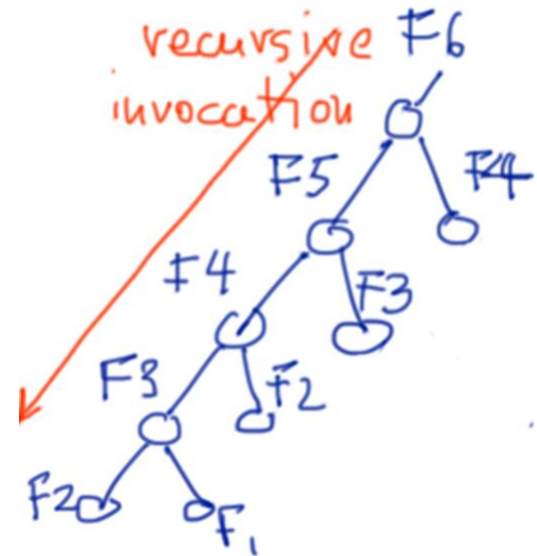
DP = recursion + re-use (**Memoization**)

Two approaches in Dynamic Programming

1. Top-down approach:

If solution is **stored** in the array,
return it (**memoization**).

Otherwise solves
subproblems recursively



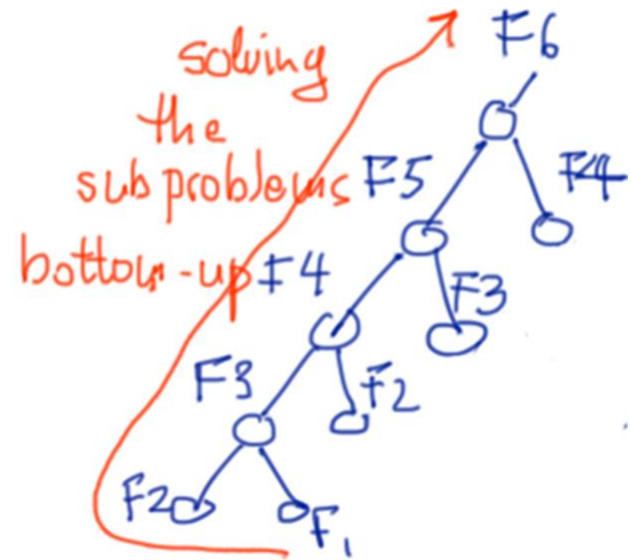
Dynamic Programming

DP = recursion + re-use (**Memoization**)

Two approaches in Dynamic Programming

2. Bottom-up approach:
Solves subproblems **iteratively**
in the order of **smallest**
to **largest sub**problems.

```
Array fib[]  
fib[1] ← 1, fib[2] ← 1  
For  $i = 3$  to  $n$  do  
     $\text{fib}[i] = \text{fib}(i - 1) + \text{fib}(i - 2)$   
return fib[n]
```



Case study I: Rod cutting problem

Problem: You are given a rod of size n and a table of prices p_1, \dots, p_n where p_i is the price in the market of a rod of size i . Determine the maximum revenue obtained by cutting the rod into pieces and selling these to the market

Example: $n = 9$,

| length i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|-------------|---|---|---|---|----|----|----|----|----|
| price p_i | 1 | 5 | 8 | 9 | 10 | 17 | 17 | 20 | 24 |

Case study I: Rod cutting problem

Problem: You are given a rod of size n and a table of prices p_1, \dots, p_n where p_i is the price in the market of a rod of size i . Determine the maximum revenue obtained by cutting the rod into pieces and selling these to the market

Example: $n = 9$,

| length i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|-------------|---|---|---|---|----|----|----|----|----|
| price p_i | 1 | 5 | 8 | 9 | 10 | 17 | 17 | 20 | 24 |

Answer: Cut the rod in two pieces, 3 and 6 and get revenue $p_3 + p_6 = 25$.

Case study I: Rod cutting problem

Problem: You are given a rod of size n and a table of prices p_1, \dots, p_n where p_i is the price in the market of a rod of size i . Determine the maximum revenue obtained by cutting the rod into pieces and selling these to the market

Example: $n = 9$,

| length i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|-------------|---|---|---|---|----|----|----|----|----|
| price p_i | 1 | 5 | 8 | 9 | 10 | 17 | 17 | 20 | 24 |

Answer: Cut the rod in two pieces, 3 and 6 and get revenue $p_3 + p_6 = 25$.

Brute force (slow): For each **possible** cut, compute the revenue and **keep** the maximum. How many possibilities? For $n = 4$, we have 1+1+1+1, 1+1+2, 1+2+1, 2+1+1, 2+2, 1+3, 3+1, 4.

Case study I: Rod cutting problem

Problem: You are given a rod of size n and a table of prices p_1, \dots, p_n where p_i is the price in the market of a rod of size i . Determine the maximum revenue obtained by cutting the rod into pieces and selling these to the market

Example: $n = 9$,

| length i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|-------------|---|---|---|---|----|----|----|----|----|
| price p_i | 1 | 5 | 8 | 9 | 10 | 17 | 17 | 20 | 24 |

Answer: Cut the rod in two pieces, 3 and 6 and get revenue $p_3 + p_6 = 25$.

Brute force (slow): For each **possible** cut, compute the revenue and **keep** the maximum. How many possibilities? For $n = 4$, we have 1+1+1+1, 1+1+2, 1+2+1, 2+1+1, 2+2, 1+3, 3+1, 4.

Exponential many 2^{n-1}
Hence exponential time

Case study I: Rod cutting problem

General Approach

Step 1: Define the problem and subproblems.

Answer: Let $DP[k]$ be the maximum value I can get from rod with size k .

Case study I: Rod cutting problem

General Approach

Step 1: Define the problem and subproblems.

Answer: Let $DP[k]$ be the maximum value I can get from rod with size k .

Step 2: Define the goal/output given Step 1.

Case study I: Rod cutting problem

General Approach

Step 1: Define the problem and subproblems.

Answer: Let $DP[k]$ be the maximum value I can get from rod with size k .

Step 2: Define the goal/output given Step 1.

It is $DP[n]$.

Case study I: Rod cutting problem

General Approach

Step 1: Define the problem and subproblems.

Answer: Let $DP[k]$ be the maximum value I can get from rod with size k .

Step 2: Define the goal/output given Step 1.

It is $DP[n]$.

Step 3: Define the base cases

Case study I: Rod cutting problem

General Approach

Step 1: Define the problem and subproblems.

Answer: Let $DP[k]$ be the maximum value I can get from rod with size k .

Step 2: Define the goal/output given Step 1.

It is $DP[n]$.

Step 3: Define the base cases

It is $DP[0] = 0$.

Step 4: Define the recurrence

Case study I: Rod cutting problem

General Approach

Step 4: Define the recurrence.

Create a recursive relationship between the subproblems (the tricky part).

Question: Given a rod of size k , where should I cut it first?

length k

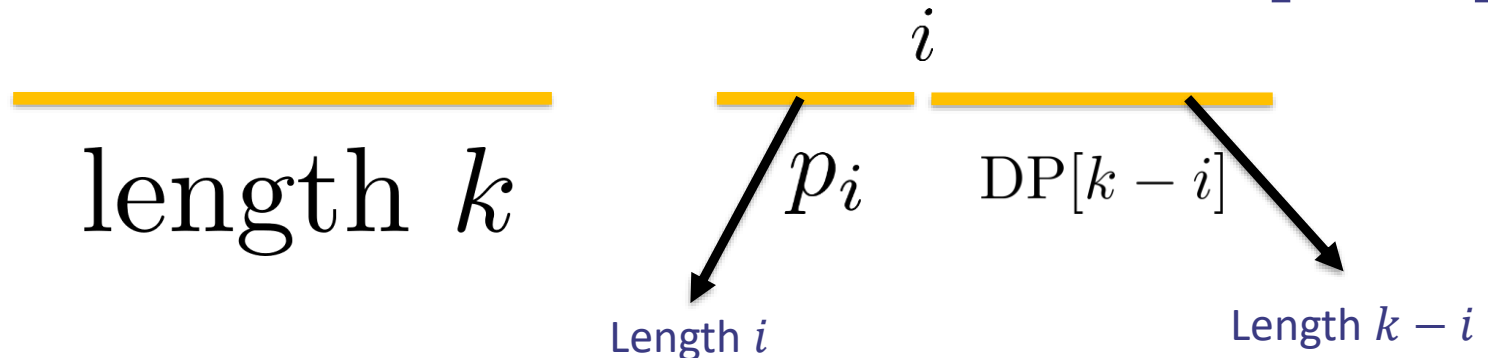
Case study I: Rod cutting problem

General Approach

Step 4: Define the recurrence.

Create a **recursive** relationship between the subproblems (the **tricky** part).

Question: Given a rod of size k , where should I cut it first? Cut at index i gives price of i and $DP[k - i]$



Case study I: Rod cutting problem

General Approach

Step 4: Define the recurrence.

Create a **recursive** relationship between the subproblems (the **tricky** part).

Question: Given a rod of size k , where should I cut it first? Cut at index i gives price of i and $DP[k - i]$



$DP[k]$ is the max of $p_i + DP[k - i]$ for all $1 \leq i \leq k$

$$DP[k] = \max_{1 \leq i \leq k} p_i + DP[k - i]$$

Case study I: Rod cutting problem

| size of piece | 1 | 2 | 3 | 4 |
|---------------|---|---|---|---|
| market price | 2 | 5 | 7 | 8 |

Rod of size $n = 4$

$DP[k]$ = maximum value from rod with size k .

$$DP[k] = \max_{1 \leq i \leq k} p_i + DP[k - i]$$

$$DP[0] = 0$$

$$DP[1] = p_1 + DP[0]$$

| | | | | |
|---|--|--|--|--|
| 0 | | | | |
|---|--|--|--|--|

$DP[0]$ $DP[1]$ $DP[2]$ $DP[3]$ $DP[4]$

Case study I: Rod cutting problem

| size of piece | 1 | 2 | 3 | 4 |
|---------------|---|---|---|---|
| market price | 2 | 5 | 7 | 8 |

Rod of size $n = 4$

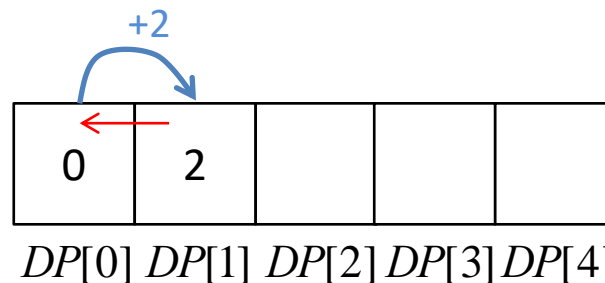
$DP[k]$ = maximum value from rod with size k .

$$DP[k] = \max_{1 \leq i \leq k} \{p_i + DP[k - i]\}$$

$$DP[0] = 0$$

$$DP[2] = \max(4, 5)$$

$$DP[1] = p_1 + DP[0] = 2$$



Case study I: Rod cutting problem

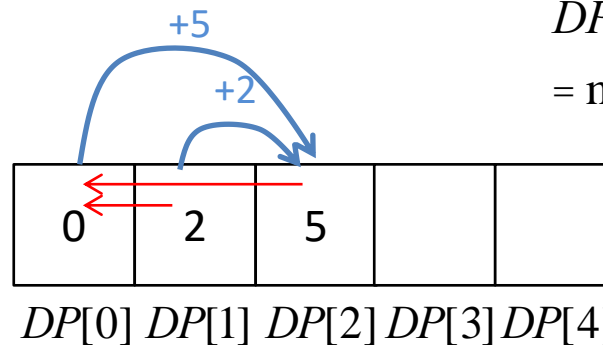
| size of piece | 1 | 2 | 3 | 4 |
|---------------|---|---|---|---|
| market price | 2 | 5 | 7 | 8 |

Rod of size $n = 4$

$DP[k]$ = maximum value from rod with size k .

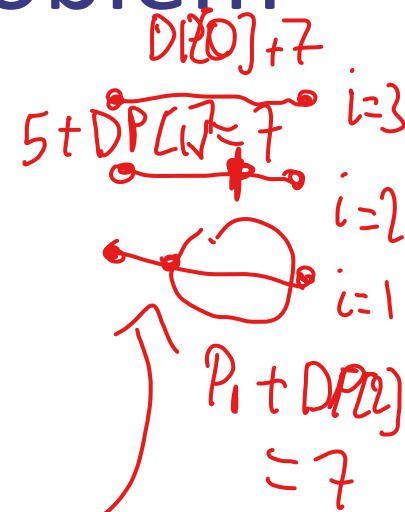
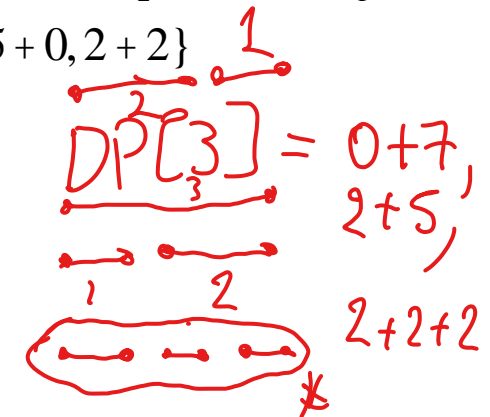
$$DP[k] = \max_{1 \leq i \leq k} \{p_i + DP[k - i]\}$$

$$DP[0] = 0$$



$$DP[2] = \max\{p_2 + DP[0], p_1 + DP[1]\}$$

$$= \max\{5 + 0, 2 + 2\}$$



Case study I: Rod cutting problem

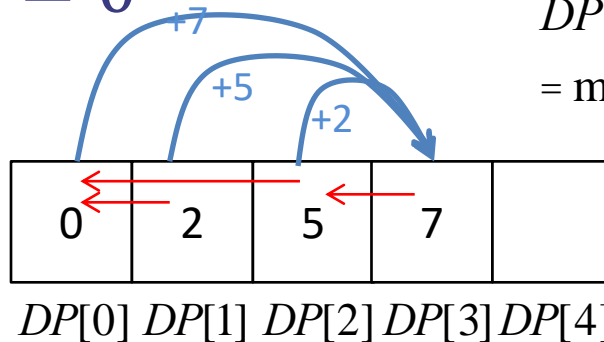
| size of piece | 1 | 2 | 3 | 4 |
|---------------|---|---|---|---|
| market price | 2 | 5 | 7 | 8 |

Rod of size $n = 4$

$DP[k]$ = maximum value from rod with size k .

$$DP[k] = \max_{1 \leq i \leq k} \{p_i + DP[k - i]\}$$

$$DP[0] = 0$$



$$\begin{aligned} DP[3] &= \max\{p_3 + DP[0], p_2 + DP[1], p_1 + DP[2]\} \\ &= \max\{7 + 0, 5 + 2, 2 + 5\} \end{aligned}$$

Case study I: Rod cutting problem

| size of piece | 1 | 2 | 3 | 4 |
|---------------|---|---|---|---|
| market price | 2 | 5 | 7 | 8 |

Rod of size $n = 4$

$DP[k]$ = maximum value from rod with size k .

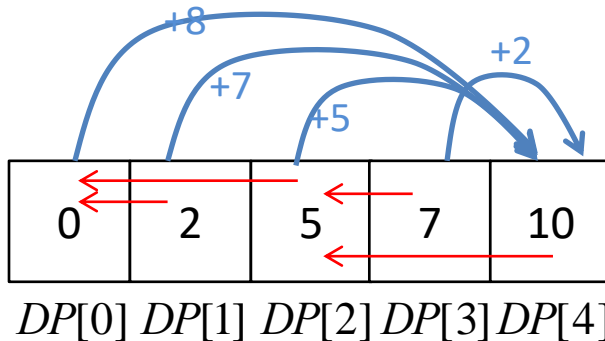
$$DP[k] = \max_{1 \leq i \leq k} \{p_i + DP[k - i]\}$$

$$DP[0] = 0$$

$$DP[4] =$$

$$\max\{p_4 + DP[0], p_3 + DP[1], p_2 + DP[2], p_1 + DP[3]\}$$

$$= \max\{8 + 0, 7 + 2, 5 + 5, 2 + 7\}$$



Case study I: Rod cutting problem

Pseudocode:

Array $DP[]$, $S[]$

$DP[0] \leftarrow 0$

For $k = 1$ to n **do**

$\max \leftarrow 0$

For $i = 1$ to k **do**

If $\max < p[i] + DP[k - i]$ **then**

$\max \leftarrow p[i] + DP[k - i]$

$DP[k] \leftarrow \max$

return $DP[n]$

Case study I: Rod cutting problem

Pseudocode:

Array $DP[]$, $S[]$

$DP[0] \leftarrow 0$

For $k = 1$ to n **do**

$\max \leftarrow 0$

For $i = 1$ to k **do**

If $\max < p[i] + DP[k - i]$ **then**

$\max \leftarrow p[i] + DP[k - i]$

$DP[k] \leftarrow \max$

return $DP[n]$

Base case

Implement recursive
formula with double
for-loop

GOAL

Running time: $\Theta(n^2)$

Case study I: Rod cutting problem

Pseudocode:

Array $DP[]$, $S[]$

$DP[0] \leftarrow 0$

For $k = 1$ to n **do**

$\max \leftarrow 0$

For $i = 1$ to k **do**

If $\max < p[i] + DP[k - i]$ **then**

$\max \leftarrow p[i] + DP[k - i]$

$DP[k] \leftarrow \max$

return $DP[n]$

Base case

Implement recursive
formula with double
for-loop

GOAL

Question: What is the cut that gives maximum revenue?

Case study I: Rod cutting problem

Pseudocode:

Array $DP[]$, $S[]$

$DP[0] \leftarrow 0$

For $k = 1$ to n **do**

$\max \leftarrow 0$

For $i = 1$ to k **do**

If $\max < p[i] + DP[k - i]$ **then**

$\max \leftarrow p[i] + DP[k - i]$

$S[k] \leftarrow i$

$DP[k] \leftarrow \max$

return $DP[n]$

Base case

Implement recursive
formula with double
for-loop

GOAL

Answer: Use pointer S

Case study I: Rod cutting problem

Example: $n = 9$

| | | | | | | | | | |
|-------------|---|---|---|---|----|----|----|----|----|
| length i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| price p_i | 1 | 5 | 8 | 9 | 10 | 17 | 17 | 20 | 24 |

| len | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|------|---|---|---|---|----|----|----|----|----|----|
| DP[] | 0 | 1 | 5 | 8 | 10 | 13 | 17 | 18 | 22 | 25 |
| S[] | 0 | 0 | 0 | 0 | 2 | 2 | 0 | 1 | 2 | 3 |

Solution for $n = 9$:

Need to cut at $S[9] = 3$. Then remaining length is $9-3=6$.

Need to cut at $S[6] = 0$. The solution is $3+6$ which give $8+17=25$

Case study I: Rod cutting problem

Example: $n = 9$

| | | | | | | | | | |
|-------------|---|---|---|---|----|----|----|----|----|
| length i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| price p_i | 1 | 5 | 8 | 9 | 10 | 17 | 17 | 20 | 24 |

| len | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|------|---|---|---|---|----|----|----|----|----|----|
| DP[] | 0 | 1 | 5 | 8 | 10 | 13 | 17 | 18 | 22 | 25 |
| S[] | 0 | 0 | 0 | 0 | 2 | 2 | 0 | 1 | 2 | 3 |

Solution for $n = 9$:

Need to cut at $S[9] = 3$. Then remaining length is $9-3=6$.

Need to cut at $S[6] = 0$. The solution is $3+6$ which give $8+17=25$

Solution for $n = 5$:

Need to cut at $S[5] = 2$. Then remaining length is $5-2=3$.

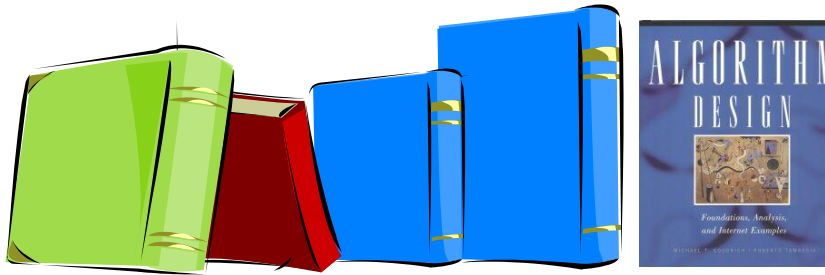
Need to cut at $S[3] = 0$. The solution is $2+3$ which give $5+8=13$

Case study II: 0/1 Knapsack

Problem: A set of n items, with each item i having positive weight w_i and positive benefit v_i . You are asked to choose items with **maximum total benefit** so that the **total weight** is **at most W**

Example:

Items:



| | | | | | |
|----------|-------|-------|-------|-------|-------|
| Weight: | 4 lbs | 2 lbs | 2 lbs | 6 lbs | 2 lbs |
| Benefit: | \$20 | \$3 | \$6 | \$25 | \$80 |

“knapsack” with 9 lbs capacity



Solution:

- item 5 (\$80, 2 lbs)
- item 3 (\$6, 2lbs)
- item 1 (\$20, 4lbs)

Case study II: 0/1 Knapsack

Idea: Dynamic Programming (first attempt).

Step 1: Define the problem and subproblems.

Answer: Let $DP[k]$ be the **maximum value** I can get from items $\{1, \dots, k\}$ without exceeding W .

Step 2: Define the goal/output given Step 1.

It is $DP[n]$.

Step 3: Define the base cases

It is $DP[0] = 0$.

Step 4: Define the recurrence

Case study II: 0/1 Knapsack

Idea: Dynamic Programming (first attempt).

Step 4: Define the recurrence

Item k will be used or not.

$$DP[k] = \max(DP[k-1], DP[k-1] + v_k)$$

But how do we know that $DP[k-1]$ does **not exceed** $W - w_k$ in weight so we can use k ?

Case study II: 0/1 Knapsack

Idea: Dynamic Programming (correct attempt).

Step 1: Define the problem and subproblems.

Answer: Let $DP[k, j]$ be the **maximum value** I can get from items $\{1, \dots, k\}$ without exceeding j .

Case study II: 0/1 Knapsack

Idea: Dynamic Programming (correct attempt).

Step 1: Define the problem and subproblems.

Answer: Let $DP[k, j]$ be the **maximum value** I can get from items $\{1, \dots, k\}$ without exceeding j .

Step 2: Define the goal/output given Step 1.

It is $DP[n, W]$.

Case study II: 0/1 Knapsack

Idea: Dynamic Programming (correct attempt).

Step 1: Define the problem and subproblems.

Answer: Let $DP[k, j]$ be the **maximum value** I can get from items $\{1, \dots, k\}$ without exceeding j .

Step 2: Define the goal/output given Step 1.

It is $DP[n, W]$.

Step 3: Define the base cases

It is $DP[0, j] = 0$ for all j and $DP[i, 0] = 0$ for all i .

Step 4: Define the recurrence

Case study II: 0/1 Knapsack

Idea: Dynamic Programming (correct attempt).

Step 4: Define the recurrence

Item k will be **used** or **not**.

$$DP[k][j] = \max(DP[k-1][j-w_k] + v_k, DP[k-1][j])$$

Case study II: 0/1 Knapsack

Idea: Dynamic Programming (correct attempt).

Step 4: Define the recurrence

Item k will be **used** or **not**.

$$DP[k][j] = \max(DP[k-1][j-w_k] + v_k, DP[k-1][j])$$

Question: How do we know that item k does not have weight more than j ?

Case study II: 0/1 Knapsack

Idea: Dynamic Programming (correct attempt).

Step 4: Define the recurrence

Item k will be **used** or **not**.

$$DP[k][j] = \begin{cases} \text{if } w_k \leq j & \max(DP[k-1][j-w_k] + v_k, DP[k-1][j]) \\ \text{If } w_k > j & DP[k-1][j] \end{cases}$$

Answer: Add an if statement in the recurrence.

Case study II: 0/1 Knapsack

Example: 3 items, $W = 4$
 $w_1 = 2, v_1 = 1, w_2 = 2, v_2 = 1, w_3 = 3, v_3 = 5$

Initialization:

| | j=0 | 1 | 2 | 3 | 4 |
|-----|-----|---|---|---|---|
| i=0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | | | | |
| 2 | 0 | | | | |
| 3 | 0 | | | | |

Case study II: 0/1 Knapsack

Example: 3 items, $W = 4$
 $w_1 = 2, v_1 = 1, w_2 = 2, v_2 = 1, w_3 = 3, v_3 = 5$

| | j=0 | 1 | 2 | 3 | 4 |
|-----|-----|-----------------|---|---|---|
| i=0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 ($j < w_1$) | | | |
| 2 | 0 | 0 ($j < w_2$) | | | |
| 3 | 0 | 0 ($j < w_3$) | | | |

Case study II: 0/1 Knapsack

Example: 3 items, $W = 4$
 $w_1 = 2, v_1 = 1, w_2 = 2, v_2 = 1, w_3 = 3, v_3 = 5$

| | j=0 | 1 | 2 | 3 | 4 |
|-----|-----|---|--------------------|---|---|
| i=0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | $\max(0, v_1 + 0)$ | | |
| 2 | 0 | 0 | | | |
| 3 | 0 | 0 | | | |

Case study II: 0/1 Knapsack

Example: 3 items, $W = 4$
 $w_1 = 2, v_1 = 1, w_2 = 2, v_2 = 1, w_3 = 3, v_3 = 5$

| | j=0 | 1 | 2 | 3 | 4 |
|-----|-----|---|------------------|---|---|
| i=0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | | |
| 2 | 0 | 0 | $\max(1, v_2+0)$ | | |
| 3 | 0 | 0 | | | |

Case study II: 0/1 Knapsack

Example: 3 items, $W = 4$
 $w_1 = 2, v_1 = 1, w_2 = 2, v_2 = 1, w_3 = 3, v_3 = 5$

| | j=0 | 1 | 2 | 3 | 4 |
|-----|-----|---|-----------------|---|---|
| i=0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | | |
| 2 | 0 | 0 | 1 | | |
| 3 | 0 | 0 | 1 ($j < w_3$) | | |

Case study II: 0/1 Knapsack

Example: 3 items, $W = 4$
 $w_1 = 2, v_1 = 1, w_2 = 2, v_2 = 1, w_3 = 3, v_3 = 5$

| | j=0 | 1 | 2 | 3 | 4 |
|-----|-----|---|---|--------------------|---|
| i=0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | $\max(0, v_1 + 0)$ | |
| 2 | 0 | 0 | 1 | | |
| 3 | 0 | 0 | 1 | | |

The table represents the dynamic programming table for the 0/1 Knapsack problem. The rows represent items (0 to 3) and the columns represent the knapsack capacity (0 to 4). The values in the table are the maximum value that can be achieved with a given capacity and a subset of items. The red arrow indicates the optimal solution for capacity 1, which is achieved by not taking any items (i=0, j=1). The green arrow indicates the optimal solution for capacity 2, which is achieved by taking item 1 (i=1, j=2).

Case study II: 0/1 Knapsack

Example: 3 items, $W = 4$
 $w_1 = 2, v_1 = 1, w_2 = 2, v_2 = 1, w_3 = 3, v_3 = 5$

| | j=0 | 1 | 2 | 3 | 4 |
|-----|-----|---|---|------------------|---|
| i=0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 | |
| 2 | 0 | 0 | 1 | $\max(1, v_2+0)$ | |
| 3 | 0 | 0 | 1 | | |

Case study II: 0/1 Knapsack

Example: 3 items, $W = 4$
 $w_1 = 2, v_1 = 1, w_2 = 2, v_2 = 1, w_3 = 3, v_3 = 5$

| | j=0 | 1 | 2 | 3 | 4 |
|-----|-----|---|---|------------------|---|
| i=0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 | |
| 2 | 0 | 0 | 1 | 1 | |
| 3 | 0 | 0 | 1 | $\max(1, v_3+0)$ | |

Case study II: 0/1 Knapsack

Example: 3 items, $W = 4$
 $w_1 = 2, v_1 = 1, w_2 = 2, v_2 = 1, w_3 = 3, v_3 = 5$

| | j=0 | 1 | 2 | 3 | 4 |
|-----|-----|---|---|---|------------------|
| i=0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 | $\max(0, v_1+0)$ |
| 2 | 0 | 0 | 1 | 1 | |
| 3 | 0 | 0 | 1 | 5 | |

Case study II: 0/1 Knapsack

Example: 3 items, $W = 4$
 $w_1 = 2, v_1 = 1, w_2 = 2, v_2 = 1, w_3 = 3, v_3 = 5$

| | j=0 | 1 | 2 | 3 | 4 |
|-----|-----|---|---|---|------------------|
| i=0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 | 1 |
| 2 | 0 | 0 | 1 | 1 | $\max(1, v_2+1)$ |
| 3 | 0 | 0 | 1 | 5 | |

Case study II: 0/1 Knapsack

Example: 3 items, $W = 4$
 $w_1 = 2, v_1 = 1, w_2 = 2, v_2 = 1, w_3 = 3, v_3 = 5$

| | j=0 | 1 | 2 | 3 | 4 |
|-----|-----|---|---|---|------------------|
| i=0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 | 1 |
| 2 | 0 | 0 | 1 | 1 | 2 |
| 3 | 0 | 0 | 1 | 5 | $\max(2, 0+v_3)$ |

Case study II: 0/1 Knapsack

Example: 3 items, $W = 4$
 $w_1 = 2, v_1 = 1, w_2 = 2, v_2 = 1, w_3 = 3, v_3 = 5$

| | j=0 | 1 | 2 | 3 | 4 |
|-----|-----|---|---|---|---|
| i=0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 | 1 |
| 2 | 0 | 0 | 1 | 1 | 2 |
| 3 | 0 | 0 | 1 | 5 | 5 |

Case study II: 0/1 Knapsack

Pseudocode:

Array $DP[][]$

For $i = 0$ to n **do**

$DP[i, 0] \leftarrow 0$

For $j = 1$ to W **do**

$DP[0, j] \leftarrow 0$

For $i = 1$ to n **do**

For $j = 1$ to W **do**

If $j < w_i$ **then**

$DP[i][j] \leftarrow DP[i - 1][j]$

else $DP[i][j] \leftarrow \max(DP[i - 1][j], DP[i - 1][j - w_i] + v_i)$

return $DP[n][W]$

Initialization

Bottom up filling DP

Goal

Case study II: 0/1 Knapsack

Pseudocode:

```
Array DP[][]
For  $i = 0$  to  $n$  do
     $DP[i, 0] \leftarrow 0$ 
For  $j = 1$  to  $W$  do
     $DP[0, j] \leftarrow 0$ 
For  $i = 1$  to  $n$  do
    For  $j = 1$  to  $W$  do
        If  $j < w_i$  then
             $DP[i][j] \leftarrow DP[i - 1][j]$ 
        else  $DP[i][j] \leftarrow \max(DP[i - 1][j], DP[i - 1][j - w_i] + v_i)$ 
return  $DP[n][W]$ 
```

Initialization

Bottom up filling DP

Goal

Running time: $\Theta(nW)$