

# Dynamic Programming

---

What is Dynamic Programming?

# Dynamic Programming

---

## What is Dynamic Programming?

- Solving a problem by divide-and-conquer implies dividing a large problem into sub-problems and solving the large problem by combining the solutions of the smaller problems.
- Dynamic Programming is like divide-and-conquer but when the sub-problems overlap.
- Dynamic Programming optimizes the solution on situations where the same sub-problems happen to be solved several times.

# Dynamic Programming

---

## What is Dynamic Programming? (History)

- A technique introduced by Richard Bellman
- Programming stands for planning not software coding as such
- Dynamic stands for flexibility or something...
- Dynamic Programming sounded as sophisticated title to get research funding :-)
- Yet the technique has shown to be a success!

# Dynamic Programming

---

## Dynamic Programming Steps

1. Express the structure of the optimal solution
2. Recursively define the value of an optimal solution
3. Compute the optimal value of an optimal solution (B-U or T-D)
4. Construct an optimal solution (when distinguish between the final solution and the computation needed to find the solution). Subtile!

# Dynamic Programming

---

## Dynamic Programming Characteristic

### 1. Optimal substructure

Optimal of the whole is obtained from optimal of the sub-problems

### 2. Overlapping sub-problems

Same sub-problems need to be solved over and over

# Problems to Explore

---

1. Fibonacci numbers
2. Rod cutting
3. Matrix Chain Multiplication
4. Knapsack
5. Activity Selection

# Fibonacci Numbers

---

$$F(n) = F(n-1) + F(n-2)$$

- Optimal substructure
- Overlapping subproblems
- TOP-Down with Memoization
- Bottom-UP following dependencies of subproblems

(Solved on blackboard)

# Rod Cutting

## Problem

Given a rod of length  $n$  and table of prices  $p_i$  where  $i = 1, 2, \dots, n$ .


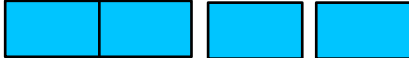

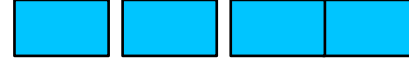




Determine the maximum revenue  $r_n$  obtain by cutting up the rod and selling the pieces.

Note!

Cutting does not cost!

## Example

Length $i$	1	2	3	4
Price ( $P_i$ )	1	5	8	9

	4
	7
	7
	7
	9
	10
	9
	9



# Rod Cutting

## The structure

The maximum revenue  $r_n$  that can be obtained for a piece of length  $n$ , is given by:

$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \dots, r_{n-1} + r_1)$$

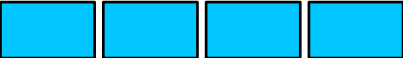




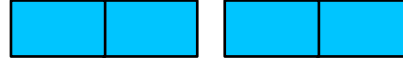


Optimal substructure because:

Solving the problem  $\max r_n$  requires solving the subproblems  $\max$  of

$$r_{n-1}, r_{n-2}, \dots, r_1$$

## Example

Length i	1	2	3	4
Price (P i)	1	5	8	9

	4
	7
	7
	7
	9
	10
	9
	9

# Rod Cutting

## Recursive definition


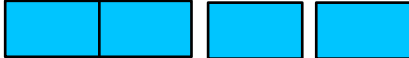

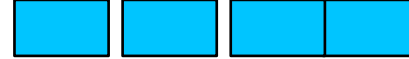




The maximum revenue  $r_n$  that can be obtained for a piece of length  $n$ , is given by:

$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \dots, r_{n-1} + r_1)$$

$$r_n = \max_{1 \leq i \leq n} (p_i + r_{n-i}) \text{ and } r_0 = 0$$

## Example

Length i	1	2	3	4
Price (P i)	1	5	8	9

	4
	7
	7
	7
	9
	10
	9
	9

# Rod Cutting

## Recursive top-down

CUT-ROD( $p, n$ )

```

1  if  $n == 0$ 
2      return 0
3   $q = -\infty$ 
4  for  $i = 1$  to  $n$ 
5       $q = \max(q, p[i] + \text{CUT-ROD}(p, n - i))$ 
6  return  $q$ 
```

$$r_n = \max_{1 \leq j \leq n} (p_i + r_{n-i}) \text{ and } r_0 = 0$$

$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \dots, r_{n-1} + r_1)$$

## Recursive top-down with memoization

(Because overlapping sub-problems)

MEMOIZED-CUT-ROD( $p, n$ )

```

1  let  $r[0..n]$  be a new array
2  for  $i = 0$  to  $n$ 
3       $r[i] = -\infty$ 
4  return MEMOIZED-CUT-ROD-AUX( $p, n, r$ )
```

MEMOIZED-CUT-ROD-AUX( $p, n, r$ )

```

1  if  $r[n] \geq 0$ 
2      return  $r[n]$ 
3  if  $n == 0$ 
4       $q = 0$ 
5  else  $q = -\infty$ 
6      for  $i = 1$  to  $n$ 
7           $q = \max(q, p[i] + \text{MEMOIZED-CUT-ROD-AUX}(p, n - i, r))$ 
8   $r[n] = q$ 
9  return  $q$ 
```

# Rod Cutting

## Top-down with memoization

MEMOIZED-CUT-ROD( $p, n$ )

```

1  let  $r[0..n]$  be a new array
2  for  $i = 0$  to  $n$ 
3       $r[i] = -\infty$ 
4  return MEMOIZED-CUT-ROD-AUX( $p, n, r$ )

```

MEMOIZED-CUT-ROD-AUX( $p, n, r$ )

```

1  if  $r[n] \geq 0$ 
2      return  $r[n]$ 
3  if  $n == 0$ 
4       $q = 0$ 
5  else  $q = -\infty$ 
6      for  $i = 1$  to  $n$ 
7           $q = \max(q, p[i] + \text{MEMOIZED-CUT-ROD-AUX}(p, n - i, r))$ 
8   $r[n] = q$ 
9  return  $q$ 

```

## Bottom-up

BOTTOM-UP-CUT-ROD( $p, n$ )

```

1  let  $r[0..n]$  be a new array
2   $r[0] = 0$ 
3  for  $j = 1$  to  $n$ 
4       $q = -\infty$ 
5      for  $i = 1$  to  $j$ 
6           $q = \max(q, p[i] + r[j - i])$ 
7       $r[j] = q$ 
8  return  $r[n]$ 

```

Solves subproblems of sizes  
 $j = 0, 1, \dots, n$  in that order

# Rod Cutting

## Construct the optimal solution

EXTENDED-BOTTOM-UP-CUT-ROD( $p, n$ )

```

1  let  $r[0..n]$  and  $s[0..n]$  be new arrays
2   $r[0] = 0$ 
3  for  $j = 1$  to  $n$ 
4       $q = -\infty$ 
5      for  $i = 1$  to  $j$ 
6          if  $q < p[i] + r[j - i]$ 
7               $q = p[i] + r[j - i]$ 
8               $s[j] = i$ 
9   $r[j] = q$ 
10 return  $r$  and  $s$ 
```

## Print the solution

PRINT-CUT-ROD-SOLUTION( $p, n$ )

```

1  ( $r, s$ ) = EXTENDED-BOTTOM-UP-CUT-ROD( $p, n$ )
2  while  $n > 0$ 
3      print  $s[n]$ 
4       $n = n - s[n]$ 
```

length $i$	1	2	3	4	5	6	7	8	9	10
price $p_i$	1	5	8	9	10	17	17	20	24	30

Solving for  $n=10$  gives this table

$i$	0	1	2	3	4	5	6	7	8	9	10
$r[i]$	0	1	5	8	10	13	17	18	22	25	30
$s[i]$	0	1	2	3	2	2	6	1	2	3	10

What would print ( $p, 5$ ) output?

# Matrix-chain multiplication

---

# Matrix-chain multiplication

---

## Problem

Given a sequence (chain)  
 $\langle A_1, A_2, \dots, A_n \rangle$ . Place  
parenthesis such the cost of  
multiplication is minimum.

## Example

Given a sequence of matrices  
 $\langle A_1 A_2 A_3 A_4 \rangle$ . Matrix multiplication is  
associative, i.e., the product can be  
obtained in 5 distinct ways:

- $(A_1(A_2(A_3A_4)))$
- $(A_1((A_2A_3)A_4))$
- $((A_1A_2)(A_3A_4))$
- $((A_1(A_2A_3))A_4)$
- $((((A_1A_2)A_3)A_4))$

# Matrix-chain multiplication

---

## Problem

If:  $A$  is  $p \times q$  matrix and  $B$  is  $q \times r$  matrix.

Then  $C = A \times B$  is  $p \times r$  matrix.

The cost of computing  $C$  is thus dominated by  $p \cdot q \cdot r$

The size of  $C$  will be  $p \cdot r$

## Example

Given a sequence of matrices  $\langle A_1, A_2, A_3 \rangle$ , where  $A_1$  is  $10 \times 100$ ,  $A_2$  is  $100 \times 5$ , and  $A_3$  is  $5 \times 50$ . What are the possible multiplication costs?

- Answer:



# Matrix-chain multiplication

## Problem

Given a sequence (chain)  
 $\langle A_1, A_2, \dots, A_n \rangle$ . A matrix  $A_i$  has  
 dimension  $p_{i-1} \times p_i$   
 The number of possible  
 parenthesizations  $P(n)$  is given by a  
 recurrent split  $k$  of parenthesised sub  
 products.

$$P(n) = \begin{cases} 1 & \text{if } n = 1, \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2. \end{cases}$$

## Example

- $(A_1(A_2(A_3A_4)))$
- $(A_1((A_2A_3)A_4))$
- $((A_1A_2)(A_3A_4))$
- $((A_1(A_2A_3))A_4)$
- $((A_1A_2)A_3)A_4$

Reason about  $P(k)$  and  $P(n-k)$

# Matrix-chain multiplication

## Optimal substructure (Step 1)

- $A_{i..j}$ , where  $i \leq j$  is the matrix dimension for the product  $A_i A_{i+1} \dots A_j$
- By have also  $A_{i..j} = A_{i..k} A_{k+1..j}$  where  $i \leq k < j$
- The cost that we get from parenthesising  $A_{i..j}$  is the cost of computing  $A_{i..k}$  plus the cost  $A_{k+1..j}$ , plus the cost of multiplying them.

## Proof of optimal substructure

- Suppose the optimal is  $A_{i..j} = A_{i..k} A_{k+1..j}$
- It means that the way we find  $A_{i..k}$  must lead to optimal cost (minimum cost), because if it was a better way than  $A_{i..k}$  is not the optimal. contradiction!

The optimal solution of a problem is composed of the optimal solution of the subproblems

# Matrix-chain multiplication

## Recursive relation (Step 2)

- Let  $m[i, j]$  be the minimum number of scalar multiplications to compute  $A_{i..j}$ , where  $1 \leq i \leq j \leq n$ . The lowest cost for  $A_{1..n}$  would then be  $m[1, n]$
- Recall that the multiplication of  $A_{i..k}A_{k+1..j}$  costs  $p_{i-1}p_kp_j$  multiplications.
- Assuming a  $k$  (optimal parenthesization)  

$$m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j$$
- We don't know  $k$  but we know it is between  $i$  and  $j$ , so:

$$m[i, j] = \begin{cases} 0 & \text{if } i = j, \\ \min_{i \leq k < j} \{m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j\} & \text{if } i < j. \end{cases}$$

# Matrix-chain multiplication

## Example

$$m[i, j] = \begin{cases} 0 & \text{if } i = j, \\ \min_{i \leq k < j} \{m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j\} & \text{if } i < j. \end{cases}$$

matrix	$A_1$	$A_2$	$A_3$	$A_4$	$A_5$	$A_6$
dimension	$30 \times 35$	$35 \times 15$	$15 \times 5$	$5 \times 10$	$10 \times 20$	$20 \times 25$

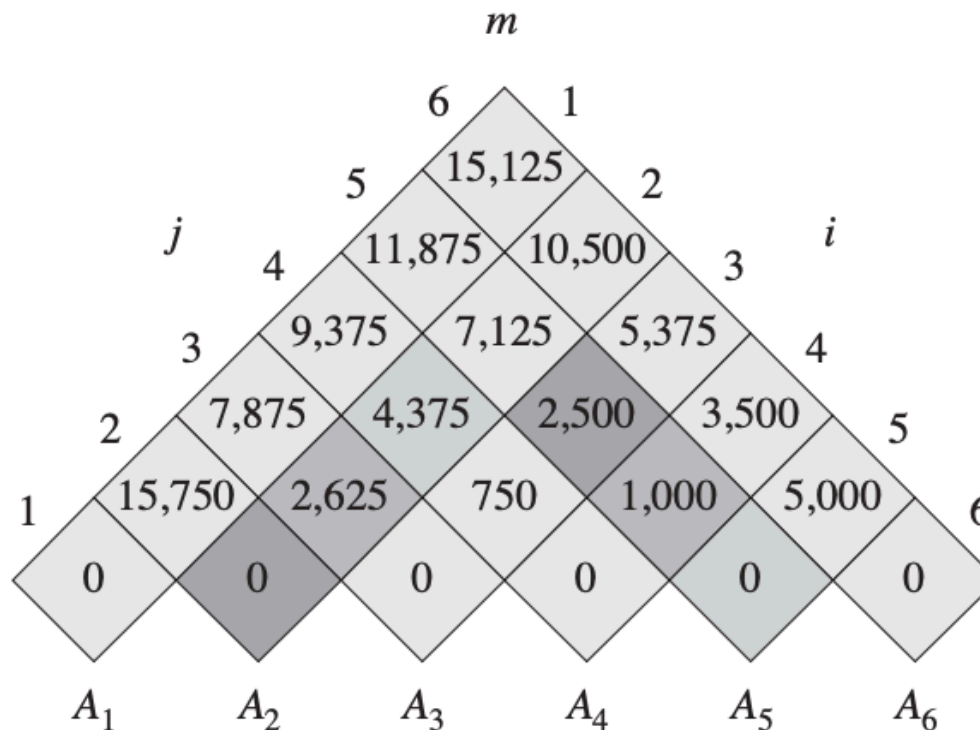
$$\begin{aligned}
 m[2, 5] &= \min \begin{cases} m[2, 2] + m[3, 5] + p_1p_2p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13,000, \\ m[2, 3] + m[4, 5] + p_1p_3p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 = 7125, \\ m[2, 4] + m[5, 5] + p_1p_4p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 = 11,375 \end{cases} \\
 &= 7125.
 \end{aligned}$$

# Matrix-chain multiplication

## Example

$$m[i, j] = \begin{cases} 0 & \text{if } i = j, \\ \min_{i \leq k < j} \{m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j\} & \text{if } i < j. \end{cases}$$

matrix	$A_1$	$A_2$	$A_3$	$A_4$	$A_5$	$A_6$
dimension	$30 \times 35$	$35 \times 15$	$15 \times 5$	$5 \times 10$	$10 \times 20$	$20 \times 25$



# Matrix-chain multiplication

## Computing the optimal costs (Step 3) Top down or bottom up

RECURSIVE-MATRIX-CHAIN( $p, i, j$ )

```

1  if  $i == j$ 
2      return 0
3   $m[i, j] = \infty$ 
4  for  $k = i$  to  $j - 1$ 
5       $q = \text{RECURSIVE-MATRIX-CHAIN}(p, i, k)$ 
        +  $\text{RECURSIVE-MATRIX-CHAIN}(p, k + 1, j)$ 
        +  $p_{i-1}p_kp_j$ 
6  if  $q < m[i, j]$ 
7       $m[i, j] = q$ 
8  return  $m[i, j]$ 

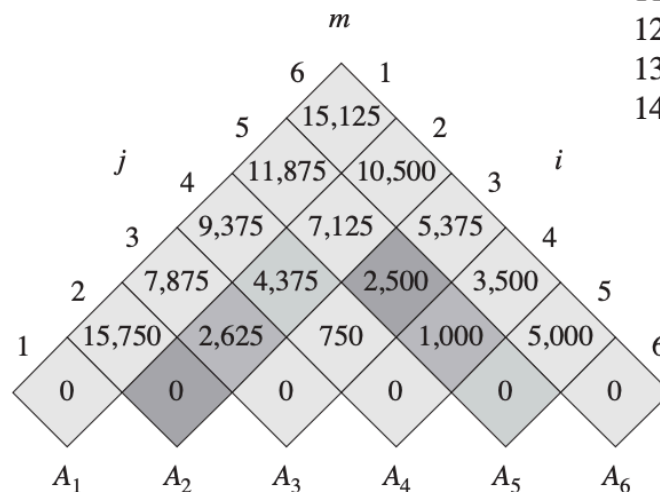
```

MATRIX-CHAIN-ORDER( $p$ )

```

1   $n = p.length - 1$ 
2  let  $m[1..n, 1..n]$  and  $s[1..n-1, 2..n]$  be new tables
3  for  $i = 1$  to  $n$ 
4       $m[i, i] = 0$ 
5  for  $l = 2$  to  $n$  //  $l$  is the chain length
6      for  $i = 1$  to  $n - l + 1$ 
7           $j = i + l - 1$ 
8           $m[i, j] = \infty$ 
9          for  $k = i$  to  $j - 1$ 
10              $q = m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j$ 
11             if  $q < m[i, j]$ 
12                  $m[i, j] = q$ 
13                  $s[i, j] = k$ 
14  return  $m$  and  $s$ 

```



# Matrix-chain multiplication

## Computing the optimal costs (Step 3) Bottom UP

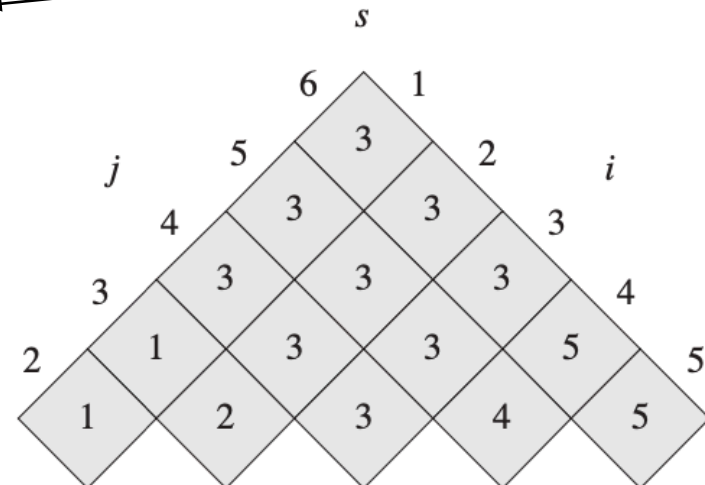
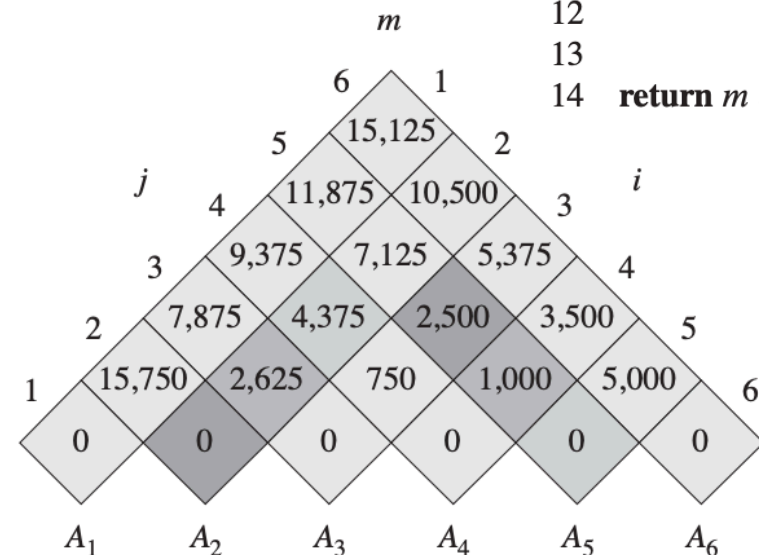
MATRIX-CHAIN-ORDER( $p$ )

```

1   $n = p.length - 1$ 
2  let  $m[1..n, 1..n]$  and  $s[1..n - 1, 2..n]$  be new tables
3  for  $i = 1$  to  $n$ 
4       $m[i, i] = 0$ 
5  for  $l = 2$  to  $n$            //  $l$  is the chain length
6      for  $i = 1$  to  $n - l + 1$ 
7           $j = i + l - 1$ 
8           $m[i, j] = \infty$ 
9          for  $k = i$  to  $j - 1$ 
10              $q = m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j$ 
11             if  $q < m[i, j]$ 
12                  $m[i, j] = q$ 
13                  $s[i, j] = k$ 
14  return  $m$  and  $s$ 
```

Each loop index takes  
at most  $n-1$

?



# Matrix-chain multiplication

## Construct the optimal solution (Step 4)

matrix	$A_1$	$A_2$	$A_3$	$A_4$	$A_5$	$A_6$
dimension	$30 \times 35$	$35 \times 15$	$15 \times 5$	$5 \times 10$	$10 \times 20$	$20 \times 25$

PRINT-OPTIMAL-PARENS( $s, i, j$ )

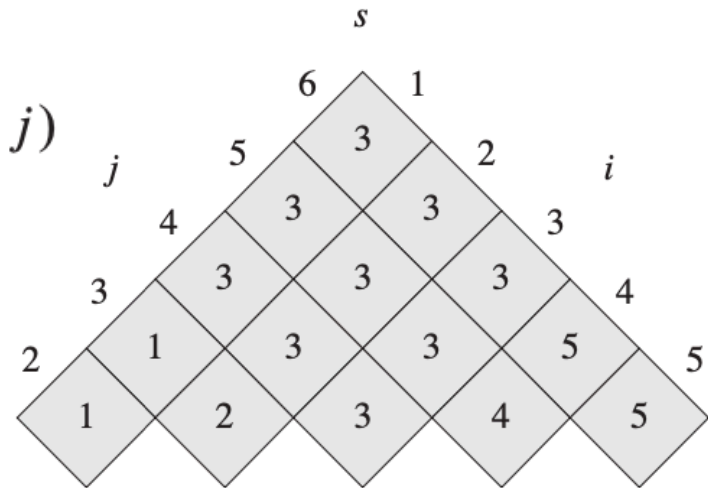
```

1  if  $i == j$ 
2      print " $A$ " $i$ 
3  else print "("
4      PRINT-OPTIMAL-PARENS( $s, i, s[i, j]$ )
5      PRINT-OPTIMAL-PARENS( $s, s[i, j] + 1, j$ )
6      print ")"

```

Calling PRINT-OPTIMAL-PARENS( $s, 1, 6$ )  
prints what?

your answer



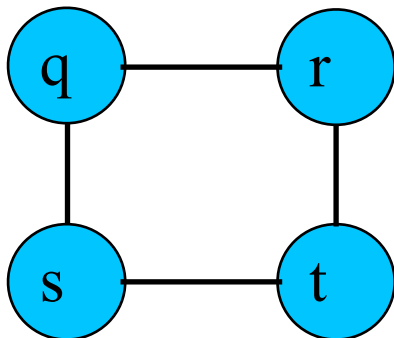


## Optimal Substructure

An optimal solution to a problem contains within it optimal solutions to subproblems such as: Rod cutting and matrix chain multiplication.

Be careful not to assume wrong!

Example: Unweighted shortest path vs Unweighted longest path



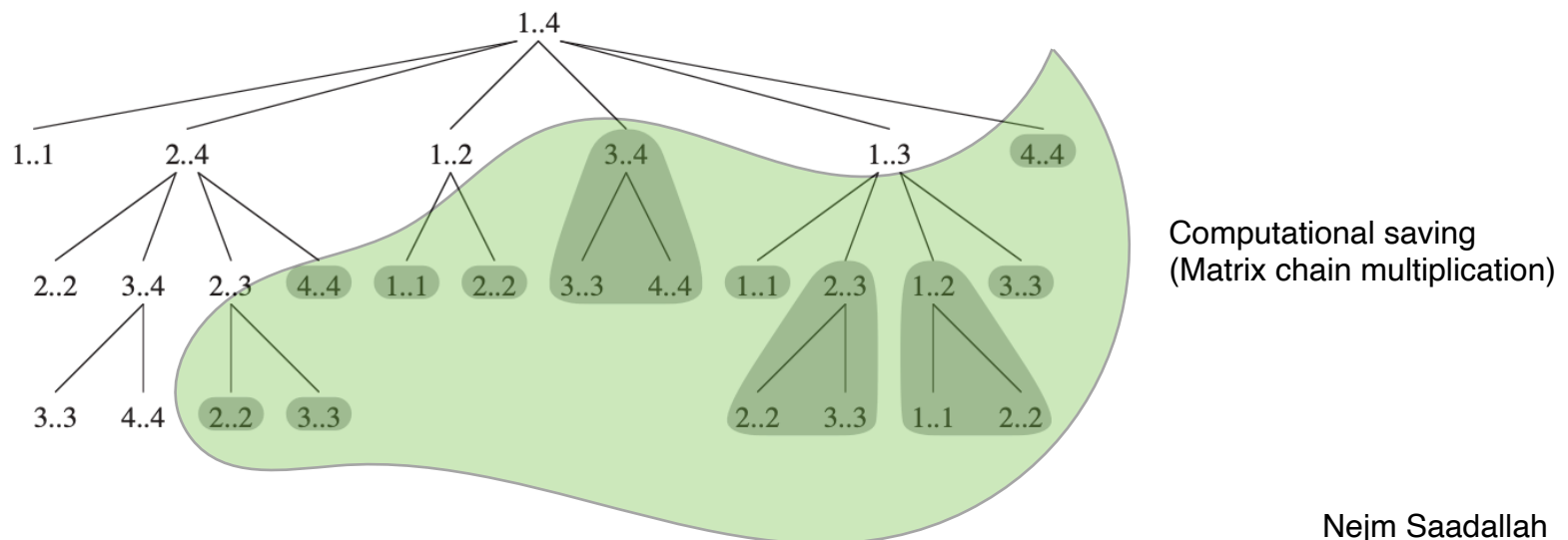
q->r->t is the longest path from q to t  
does not contain in it the longest path  
from q to r !

# Dynamic Programming Ingredients

## Overlapping subproblems

The set of subproblems is relatively small (polynomial in the input size).  
When an algorithm (often recursive) revisits the same problem repeatedly gives the problem the property of “Overlapping subproblems”

# Memoization



# Activity Selection

---

## Problem

A Set of activities  $S = \{a_1, a_2, \dots, a_n\}$  that require a resource. Each activity  $a_i$  has a start time  $s_i$  and a finish time  $f_i$ , where  $0 \leq s_i < f_i < \infty$ , if  $a_i$  is selected it takes place in the half-open interval  $[s_i, f_i)$ .

Two activities  $a_i$  and  $a_j$  are compatible if the intervals  $[s_i, f_i)$  and  $[s_j, f_j)$  do not overlap:  $s_i \geq f_j$  or  $s_j \geq f_i$ .

We want to select a maximum subset of mutually compatible activities!

# Activity Selection

## Example

$i$	1	2	3	4	5	6	7	8	9	10	11
$s_i$	1	3	0	5	3	5	6	8	8	2	12
$f_i$	4	5	6	7	9	9	10	11	12	14	16

Problem: Find the largest subset of mutually compatible activities.

Solution:

# Activity Selection

## Example

$i$	1	2	3	4	5	6	7	8	9	10	11
$s_i$	1	3	0	5	3	5	6	8	8	2	12
$f_i$	4	5	6	7	9	9	10	11	12	14	16

Problem: Find the largest subset of mutually compatible activities.

Solution:  $\{a_1, a_4, a_8, a_{11}\}$  or  $\{a_2, a_4, a_9, a_{11}\}$

# Activity Selection

## Dynamic programming approach

$S_{ij}$  the set of activities starting after  $f_i$  and finishing after  $a_j$  starts.

Example:

$$S_{1,7} = \{a_4, a_6, a_7\}$$

$i$	1	2	3	4	5	6	7	8	9	10	11
$s_i$	1	3	0	5	3	5	6	8	8	2	12
$f_i$	4	5	6	7	9	9	10	11	12	14	16

# Activity Selection

## Dynamic programming approach

- $S_{ij}$  the set activities starting after  $f_i$  and finishing after  $a_j$  starts
- Suppose  $A_{ij}$  is a maximum set of mutually compatible activities in  $S_{ij}$ , and that  $a_k \in A_{ij}$
- With  $a_k$  in an optimal solution means  $A_{ik} = A_{ij} \cap S_{ik}$  and  $A_{kj} = A_{ij} \cap S_{kj}$
- Here  $A_{ik}$  are the activities that finish before  $a_k$  starts and  $A_{kj}$  are the activities that start after  $a_k$  finishes:
- $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj} \Rightarrow$  the maximum size of  $A_{ij}$  from  $S_{ij}$  is  
 $|A_{ij}| = |A_{ik}| + |A_{kj}| + 1$

# Activity Selection

## Dynamic programming approach

- $S_{ij}$  the set activities starting after  $f_i$  and finishing after  $a_j$  starts
- Suppose  $A_{ij}$  is a maximum set of mutually compatible activities in  $S_{ij}$ , and that  $a_k \in A_{ij}$
- With  $a_k$  in an optimal solution means  $A_{ik} = A_{ij} \cap S_{ik}$  and  $A_{kj} = A_{ij} \cap S_{kj}$
- Here  $A_{ik}$  are the activities that finish before  $a_k$  starts and  $A_{kj}$  are the activities that start after  $a_k$  finishes:
- $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj} \Rightarrow$  the maximum size of  $A_{ij}$  from  $S_{ij}$  is  
 $|A_{ij}| = |A_{ik}| + |A_{kj}| + 1$       This the structure of our solution



# Activity Selection

## Dynamic programming approach

Claim:

$A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj} \Rightarrow$  the maximum size of  $A_{ij}$  from  $S_{ij}$  is

$$|A_{ij}| = |A_{ik}| + |A_{kj}| + 1$$

Direction of a Proof:

If it was  $|A'_{kj}|$  of mutually compatible activities in  $S_{kj}$  and  $|A'_{kj}| > |A_{kj}|$  then we could use  $A'_{kj}$  to construct  $|A_{ik}| + |A'_{kj}| + 1 > |A_{ik}| + |A_{kj}| + 1 = |A_{ij}|$ , but  $A_{ij}$  is an optimal solution. **Contradiction ! We must have  $|A'_{kj}| = |A_{kj}|$**

# Activity Selection

---

## Dynamic programming approach

Solution:

The size of an optimal solution for  $S_{ij}$  is  $c[i, j] = c[i, k] + c[k, j] + 1$

because we don't know  $k$ , we have to find it:

$$c[i, j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset, \\ \max_{a_k \in S_{ij}} \{c[i, k] + c[k, j] + 1\} & \text{if } S_{ij} \neq \emptyset. \end{cases} \quad (16.2)$$

# Activity Selection

## Greedy approach

$S_k = \{a_i \in S : f_k \leq s_i\}$  be the set of activities that starts after  $a_k$  finishes

### **Theorem 16.1**

Consider any nonempty subproblem  $S_k$ , and let  $a_m$  be an activity in  $S_k$  with the earliest finish time. Then  $a_m$  is included in some maximum-size subset of mutually compatible activities of  $S_k$ .

Example:

$i$	1	2	3	4	5	6	7	8	9	10	11
$s_i$	1	3	0	5	3	5	6	8	8	2	12
$f_i$	4	5	6	7	9	9	10	11	12	14	16

Solution:

# Activity Selection

## Greedy approach

$S_k = \{a_i \in S : f_k \leq s_i\}$  be the set of activities that starts after  $a_k$  finishes

### **Theorem 16.1**

Consider any nonempty subproblem  $S_k$ , and let  $a_m$  be an activity in  $S_k$  with the earliest finish time. Then  $a_m$  is included in some maximum-size subset of mutually compatible activities of  $S_k$ .

Example:

$i$	1	2	3	4	5	6	7	8	9	10	11
$s_i$	1	3	0	5	3	5	6	8	8	2	12
$f_i$	4	5	6	7	9	9	10	11	12	14	16

Solution:  $\{a_1, a_4, a_8, a_{11}\}$  or  $\{a_2, a_4, a_9, a_{11}\}$

# Activity Selection

## Greedy approach

$S_k = \{a_i \in S : f_k \leq s_i\}$  be the set of activities that starts after  $a_k$  finishes

### **Theorem 16.1**

Consider any nonempty subproblem  $S_k$ , and let  $a_m$  be an activity in  $S_k$  with the earliest finish time. Then  $a_m$  is included in some maximum-size subset of mutually compatible activities of  $S_k$ .

**Proof** Let  $A_k$  be a maximum-size subset of mutually compatible activities in  $S_k$ , and let  $a_j$  be the activity in  $A_k$  with the earliest finish time. If  $a_j = a_m$ , we are done, since we have shown that  $a_m$  is in some maximum-size subset of mutually compatible activities of  $S_k$ . If  $a_j \neq a_m$ , let the set  $A'_k = A_k - \{a_j\} \cup \{a_m\}$  be  $A_k$  but substituting  $a_m$  for  $a_j$ . The activities in  $A'_k$  are disjoint, which follows because the activities in  $A_k$  are disjoint,  $a_j$  is the first activity in  $A_k$  to finish, and  $f_m \leq f_j$ . Since  $|A'_k| = |A_k|$ , we conclude that  $A'_k$  is a maximum-size subset of mutually compatible activities of  $S_k$ , and it includes  $a_m$ . ■

# 0/1 Knapsack

---

## Problem 0/1 Knapsack

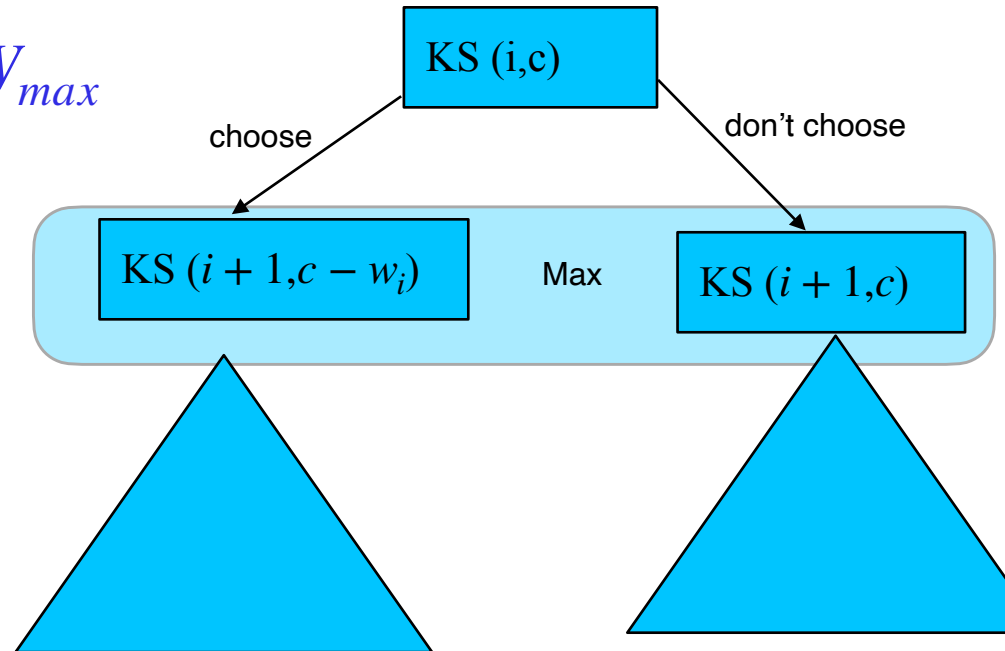
- A set of  $n$  items  $I = i_1, i_2, \dots, i_n$
- A Knapsack with maximum capacity  $W_{max}$
- Every item has a value  $v_i$  and a weight  $w_i$

Determine the maximum value  $V_{max}$  that can be obtained by selecting a subset of items such that the total weight of the items does not exceed  $W_{max}$

# 0/1 Knapsack

## Structure of the solution (optimal substructure)

- Let  $KS(i, c)$  be the maximum value that can be obtained using up to the  $i$ -th item with a total weight  $\leq c$ . ( $c$  is available capacity)
- $x_0 \cdot v_0 + x_1 \cdot v_1 + \dots x_n \cdot v_n = V_{max}$
- $x_0 \cdot w_0 + x_1 \cdot w_1 + \dots x_n \cdot w_n = W_{max}$
- $x_i \in \{0,1\}$  choosing or not



# 0/1 Knapsack

---

## 0/1 Knapsack Problem

- A set of  $n$  items  $I = i_1, i_2, \dots, i_n$
- A Knapsack with maximum capacity  $W_{max}$
- Every item has a value  $v_i$  and a weight  $w_i$

Determine the maximum value  $V_{max}$  that can be obtained by selecting a subset of items such that the total weight of the items does not exceed  $W_{max}$

Can we solve it with a greedy approach?



# Fractional Knapsack

---

## Fractional Knapsack Problem

- A set of  $n$  items  $I = i_1, i_2, \dots, i_n$
- A Knapsack with maximum capacity  $W_{max}$
- Every item has a value  $v_i$  and a weight  $w_i$

Determine the maximum value  $V_{max}$  that can be obtained by selecting a **FRACTION** from the items such that the total weight of the items does not exceed  $W_{max}$

Can we solve it with a greedy approach?

# Additional resources

---

1. [https://www.youtube.com/watch?v=r4-cftqTcdI&ab\\_channel=MITOpenCourseWare](https://www.youtube.com/watch?v=r4-cftqTcdI&ab_channel=MITOpenCourseWare)
- 2.