

Elements of Dynamic Programming

COMS10017 - Algorithms 1

Dr Christian Konrad

Solving a Problem with Dynamic Programming:

Solving a Problem with Dynamic Programming:

- 1 Identify optimal substructure

Solving a Problem with Dynamic Programming:

- ① Identify optimal substructure
- ② Give recursive solution

Solving a Problem with Dynamic Programming:

- ① Identify optimal substructure
- ② Give recursive solution
- ③ Compute optimal costs

Solving a Problem with Dynamic Programming:

- ① Identify optimal substructure
- ② Give recursive solution
- ③ Compute optimal costs
- ④ Construct optimal solution

Solving a Problem with Dynamic Programming:

- ① Identify optimal substructure
- ② Give recursive solution
- ③ Compute optimal costs
- ④ Construct optimal solution

Discussion:

Solving a Problem with Dynamic Programming:

- 1 Identify optimal substructure
- 2 Give recursive solution
- 3 Compute optimal costs
- 4 Construct optimal solution

Discussion:

- Steps 1 and 2 requires studying the problem at hand

Solving a Problem with Dynamic Programming:

- 1 Identify optimal substructure
- 2 Give recursive solution
- 3 Compute optimal costs
- 4 Construct optimal solution

Discussion:

- Steps 1 and 2 requires studying the problem at hand
- Steps 3 and 4 are usually straightforward

Step 1: Identify Optimal Substructure

Optimal Substructure

Step 1: Identify Optimal Substructure

Optimal Substructure Problem **P** exhibits *optimal substructure* if:

Step 1: Identify Optimal Substructure

Optimal Substructure Problem **P** exhibits *optimal substructure* if:

An optimal solution to **P** contains within it optimal solutions to subproblems of **P**.

Step 1: Identify Optimal Substructure

Optimal Substructure Problem \mathbf{P} exhibits *optimal substructure* if:

An optimal solution to \mathbf{P} contains within it optimal solutions to subproblems of \mathbf{P} .

Examples: Let OPT be optimal solution

Step 1: Identify Optimal Substructure

Optimal Substructure Problem \mathbf{P} exhibits *optimal substructure* if:

An optimal solution to \mathbf{P} contains within it optimal solutions to subproblems of \mathbf{P} .

Examples: Let OPT be optimal solution

- POLE-CUTTING:

Step 1: Identify Optimal Substructure

Optimal Substructure Problem **P** exhibits *optimal substructure* if:

An optimal solution to **P** contains within it optimal solutions to subproblems of **P**.

Examples: Let *OPT* be optimal solution

- **POLE-CUTTING:** If *OPT* cuts at position k then cuts within $\{1, \dots, k-1\}$ form opt. solution to pole of len. k , and cuts within $\{k+1, \dots, n\}$ form opt. solution to pole of len. $n-k$.



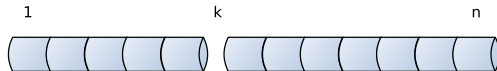
Step 1: Identify Optimal Substructure

Optimal Substructure Problem \mathbf{P} exhibits *optimal substructure* if:

An optimal solution to \mathbf{P} contains within it optimal solutions to subproblems of \mathbf{P} .

Examples: Let OPT be optimal solution

- **POLE-CUTTING:** If OPT cuts at position k then cuts within $\{1, \dots, k-1\}$ form opt. solution to pole of len. k , and cuts within $\{k+1, \dots, n\}$ form opt. solution to pole of len. $n-k$.



- **MATRIX-CHAIN-PARENTHEZIZATION:**

Step 1: Identify Optimal Substructure

Optimal Substructure Problem **P** exhibits *optimal substructure* if:

An optimal solution to **P** contains within it optimal solutions to subproblems of **P**.

Examples: Let *OPT* be optimal solution

- **POLE-CUTTING:** If *OPT* cuts at position k then cuts within $\{1, \dots, k-1\}$ form opt. solution to pole of len. k , and cuts within $\{k+1, \dots, n\}$ form opt. solution to pole of len. $n-k$.



- **MATRIX-CHAIN-PARENTHEZIZATION:** If in *OPT* final multiplication is $A_{1k} \times A_{(k+1)n}$ then *OPT* contains optimal parenthesizations of $A_1 \times \dots \times A_k$ and $A_{k+1} \times \dots \times A_n$

$$(A_1 \times (A_2 \times A_3)) \times ((A_4 \times A_5) \times A_6)$$

Step 2. Give Recursive Solution

Define Table for Storing Optimal Solutions to Subproblems:

Step 2. Give Recursive Solution

Define Table for Storing Optimal Solutions to Subproblems:

Optimal substructure indicates how subproblems look like

Step 2. Give Recursive Solution

Define Table for Storing Optimal Solutions to Subproblems:

Optimal substructure indicates how subproblems look like

- POLE-CUTTING:

Step 2. Give Recursive Solution

Define Table for Storing Optimal Solutions to Subproblems:

Optimal substructure indicates how subproblems look like

- **POLE-CUTTING:**
OPT contains optimal solutions to shorter lengths

Step 2. Give Recursive Solution

Define Table for Storing Optimal Solutions to Subproblems:

Optimal substructure indicates how subproblems look like

- POLE-CUTTING:
OPT contains optimal solutions to shorter lengths
→ Store optimal solutions for every length in $\{1, \dots, n\}$
(table of length n)

Step 2. Give Recursive Solution

Define Table for Storing Optimal Solutions to Subproblems:

Optimal substructure indicates how subproblems look like

- POLE-CUTTING:
OPT contains optimal solutions to shorter lengths
→ Store optimal solutions for every length in $\{1, \dots, n\}$
(table of length n)
- MATRIX-CHAIN-PARENTHEZIZATION:

Step 2. Give Recursive Solution

Define Table for Storing Optimal Solutions to Subproblems:

Optimal substructure indicates how subproblems look like

- POLE-CUTTING:
OPT contains optimal solutions to shorter lengths
→ Store optimal solutions for every length in $\{1, \dots, n\}$
(table of length n)
- MATRIX-CHAIN-PARENTHEZIZATION:
OPT contains optimal parenthesizations for subproducts
 $A_i \times \dots \times A_j$

Step 2. Give Recursive Solution

Define Table for Storing Optimal Solutions to Subproblems:

Optimal substructure indicates how subproblems look like

- POLE-CUTTING:
OPT contains optimal solutions to shorter lengths
→ Store optimal solutions for every length in $\{1, \dots, n\}$
(table of length n)
- MATRIX-CHAIN-PARENTHEZIZATION:
OPT contains optimal parenthesizations for subproducts
 $A_i \times \dots \times A_j$
→ Store optimal parenthesizations for every subproduct

Step 2. Give Recursive Solution

Define Table for Storing Optimal Solutions to Subproblems:

Optimal substructure indicates how subproblems look like

- POLE-CUTTING:
OPT contains optimal solutions to shorter lengths
→ Store optimal solutions for every length in $\{1, \dots, n\}$
(table of length n)
- MATRIX-CHAIN-PARENTHEZIZATION:
OPT contains optimal parenthesizations for subproducts
 $A_i \times \dots \times A_j$
→ Store optimal parenthesizations for every subproduct
 $A_i \times \dots \times A_j$ (table of size n^2)

Step 2. Give Recursive Solution (2)

Express Optimal Solutions Recursively:

Step 2. Give Recursive Solution (2)

Express Optimal Solutions Recursively:

- POLE-CUTTING: (p_k : price for selling a pole of length k)

$m[i] :=$ maximum revenue to pole of length i

Step 2. Give Recursive Solution (2)

Express Optimal Solutions Recursively:

- POLE-CUTTING: (p_k : price for selling a pole of length k)

$m[i]$:= maximum revenue to pole of length i

$$m[i] = \max_{1 \leq k \leq i} p_k + m_{i-k}$$

Step 2. Give Recursive Solution (2)

Express Optimal Solutions Recursively:

- POLE-CUTTING: (p_k : price for selling a pole of length k)

$m[i] :=$ maximum revenue to pole of length i

$$m[i] = \max_{1 \leq k \leq i} p_k + m_{i-k}$$

- MATRIX-CHAIN-PARENTHEZIZATION:

Step 2. Give Recursive Solution (2)

Express Optimal Solutions Recursively:

- POLE-CUTTING: (p_k : price for selling a pole of length k)

$m[i] :=$ maximum revenue to pole of length i

$$m[i] = \max_{1 \leq k \leq i} p_k + m_{i-k}$$

- MATRIX-CHAIN-PARENTHEZIZATION:

$m[i, j] :=$ min. # scalar mult. to compute $A_i \times A_{i+1} \times \cdots \times A_j$

Step 2. Give Recursive Solution (2)

Express Optimal Solutions Recursively:

- POLE-CUTTING: (p_k : price for selling a pole of length k)

$m[i] :=$ maximum revenue to pole of length i

$$m[i] = \max_{1 \leq k \leq i} p_k + m_{i-k}$$

- MATRIX-CHAIN-PARENTHEZIZATION:

$m[i, j] :=$ min. # scalar mult. to compute $A_i \times A_{i+1} \times \cdots \times A_j$

$$\begin{aligned} m[i, j] &= \min_{i \leq k < j} m[i, k] + m[k + 1, j] \\ &+ \text{"cost for computing } A_{ik} \times A_{(k+1)j}\text{"} \end{aligned}$$

Two Possibilities:

Two Possibilities:

- Bottom-up

Two Possibilities:

- Bottom-up
- Top-down with memoization

Two Possibilities:

- Bottom-up
- Top-down with memoization

Example:

Compute Optimal Costs

Two Possibilities:

- Bottom-up
- Top-down with memoization

Example: Bottom-up for POLE-CUTTING

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

$$m[i] = \max_{1 \leq k \leq i} p_k + m_{i-k}$$

0	1	2	3	4	5	6	7	8	9	10
-	-	-	-	-	-	-	-	-	-	-

Compute Optimal Costs

Two Possibilities:

- Bottom-up
- Top-down with memoization

Example: Bottom-up for POLE-CUTTING

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

$$m[i] = \max_{1 \leq k \leq i} p_k + m_{i-k}$$

0	1	2	3	4	5	6	7	8	9	10
-	-	-	-	-	-	-	-	-	-	-

Initialize base cases: $m[0] = 0$ and $m[1] = p_1$

Compute Optimal Costs

Two Possibilities:

- Bottom-up
- Top-down with memoization

Example: Bottom-up for POLE-CUTTING

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

$$m[i] = \max_{1 \leq k \leq i} p_k + m_{i-k}$$

0	1	2	3	4	5	6	7	8	9	10
0	1	-	-	-	-	-	-	-	-	-

Initialize base cases: $m[0] = 0$ and $m[1] = p_1$

Compute Optimal Costs

Two Possibilities:

- Bottom-up
- Top-down with memoization

Example: Bottom-up for POLE-CUTTING

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

$$m[i] = \max_{1 \leq k \leq i} p_k + m_{i-k}$$

0	1	2	3	4	5	6	7	8	9	10
0	1	-	-	-	-	-	-	-	-	-

$$m[2] = \max\{p_1 + m_1, p_2 + m_0\} = \max\{1 + 1, 5 + 0\} = 5$$

Compute Optimal Costs

Two Possibilities:

- Bottom-up
- Top-down with memoization

Example: Bottom-up for POLE-CUTTING

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

$$m[i] = \max_{1 \leq k \leq i} p_k + m_{i-k}$$

0	1	2	3	4	5	6	7	8	9	10
0	1	5	-	-	-	-	-	-	-	-

$$m[2] = \max\{p_1 + m_1, p_2 + m_0\} = \max\{1 + 1, 5 + 0\} = 5$$

Compute Optimal Costs

Two Possibilities:

- Bottom-up
- Top-down with memoization

Example: Bottom-up for POLE-CUTTING

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

$$m[i] = \max_{1 \leq k \leq i} p_k + m_{i-k}$$

0	1	2	3	4	5	6	7	8	9	10
0	1	5	-	-	-	-	-	-	-	-

$$m[3] = \max\{p_1 + m_2, p_2 + m_1, p_3 + m_0\} = \max\{1 + 5, 5 + 1, 8 + 0\} = 8$$

Compute Optimal Costs

Two Possibilities:

- Bottom-up
- Top-down with memoization

Example: Bottom-up for POLE-CUTTING

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

$$m[i] = \max_{1 \leq k \leq i} p_k + m_{i-k}$$

0	1	2	3	4	5	6	7	8	9	10
0	1	5	8	-	-	-	-	-	-	-

$$m[3] = \max\{p_1 + m_2, p_2 + m_1, p_3 + m_0\} = \max\{1 + 5, 5 + 1, 8 + 0\} = 8$$

Compute Optimal Costs

Two Possibilities:

- Bottom-up
- Top-down with memoization

Example: Bottom-up for POLE-CUTTING

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

$$m[i] = \max_{1 \leq k \leq i} p_k + m_{i-k}$$

0	1	2	3	4	5	6	7	8	9	10
0	1	5	8	-	-	-	-	-	-	-

$$m[4] = \max\{p_1 + m_3, p_2 + m_2, p_3 + m_1, p_4 + m_0\} = \max\{1 + 8, 5 + 5, 8 + 1, 9\} = 10$$

Compute Optimal Costs

Two Possibilities:

- Bottom-up
- Top-down with memoization

Example: Bottom-up for POLE-CUTTING

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

$$m[i] = \max_{1 \leq k \leq i} p_k + m_{i-k}$$

0	1	2	3	4	5	6	7	8	9	10
0	1	5	8	10	-	-	-	-	-	-

$$m[4] = \max\{p_1 + m_3, p_2 + m_2, p_3 + m_1, p_4 + m_0\} = \max\{1 + 8, 5 + 5, 8 + 1, 9\} = 10$$

Compute Optimal Costs

Two Possibilities:

- Bottom-up
- Top-down with memoization

Example: Bottom-up for POLE-CUTTING

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

$$m[i] = \max_{1 \leq k \leq i} p_k + m_{i-k}$$

0	1	2	3	4	5	6	7	8	9	10
0	1	5	8	10	-	-	-	-	-	-

$$m[5] = \max\{p_1 + m_4, p_2 + m_3, p_3 + m_2, p_4 + m_1, p_5 + m_0\} = \max\{1 + 10, 5 + 8, 8 + 2, 9 + 1, 10\} = 13$$

Compute Optimal Costs

Two Possibilities:

- Bottom-up
- Top-down with memoization

Example: Bottom-up for POLE-CUTTING

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

$$m[i] = \max_{1 \leq k \leq i} p_k + m_{i-k}$$

0	1	2	3	4	5	6	7	8	9	10
0	1	5	8	10	13	-	-	-	-	-

$$m[5] = \max\{p_1 + m_4, p_2 + m_3, p_3 + m_2, p_4 + m_1, p_5 + m_0\} = \max\{1 + 10, 5 + 8, 8 + 2, 9 + 1, 10\} = 13$$

Compute Optimal Costs

Two Possibilities:

- Bottom-up
- Top-down with memoization

Example: Bottom-up for POLE-CUTTING

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

$$m[i] = \max_{1 \leq k \leq i} p_k + m_{i-k}$$

0	1	2	3	4	5	6	7	8	9	10
0	1	5	8	10	13	-	-	-	-	-

...

Compute Optimal Costs

Two Possibilities:

- Bottom-up
- Top-down with memoization

Example: Bottom-up for POLE-CUTTING

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

$$m[i] = \max_{1 \leq k \leq i} p_k + m_{i-k}$$

0	1	2	3	4	5	6	7	8	9	10
0	1	5	8	10	13	17	18	22	25	30

Compute Optimal Costs

Two Possibilities:

- Bottom-up
- Top-down with memoization

Example: Bottom-up for POLE-CUTTING

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

$$m[i] = \max_{1 \leq k \leq i} p_k + m_{i-k}$$

0	1	2	3	4	5	6	7	8	9	10
0	1	5	8	10	13	17	18	22	25	30

The maximum revenue obtainable for a pole of length 10 is 30

Compute Optimal Costs

Two Possibilities:

- Bottom-up
- Top-down with memoization

Example: Bottom-up for POLE-CUTTING

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

$$m[i] = \max_{1 \leq k \leq i} p_k + m_{i-k}$$

0	1	2	3	4	5	6	7	8	9	10
0	1	5	8	10	13	17	18	22	25	30

But how can we find out how to cut the pole?

Step 4: Construct Optimal Solution

Keep Track of Optimal Choices: store optimal choices in array s

Require: Integer n , array p of length n with prices

Let $r[0 \dots n]$ be a new array

$r[0] \leftarrow 0$

for $j = 1 \dots n$ **do**

$r[j] \leftarrow -\infty$

for $i = 1 \dots j$ **do**

$r[j] \leftarrow \max\{r[j], p[i] + r[j - i]\}$

return $r[n]$

Algorithm BOTTOM-UP-CUT-POLE(p, n)

Step 4: Construct Optimal Solution

Keep Track of Optimal Choices: store optimal choices in array s

```
Require: Integer  $n$ , array  $p$  of length  $n$  with prices  
Let  $r[0 \dots n]$  be a new array, let  $s[1 \dots n]$  be a new array  
 $r[0] \leftarrow 0$   
for  $j = 1 \dots n$  do  
   $r[j] \leftarrow -\infty$   
  for  $i = 1 \dots j$  do  
    if  $p[i] + r[j - i] > q$  then  
       $r[j] \leftarrow p[i] + r[j - i]$   
       $s[j] \leftarrow i$   
return  $r[n]$ 
```

Algorithm BOTTOM-UP-CUT-POLE(p, n)

Step 4: Construct Optimal Solution

Keep Track of Optimal Choices: store optimal choices in array s

```
Require: Integer  $n$ , array  $p$  of length  $n$  with prices  
Let  $r[0 \dots n]$  be a new array, let  $s[1 \dots n]$  be a new array  
 $r[0] \leftarrow 0$   
for  $j = 1 \dots n$  do  
   $r[j] \leftarrow -\infty$   
  for  $i = 1 \dots j$  do  
    if  $p[i] + r[j - i] > q$  then  
       $r[j] \leftarrow p[i] + r[j - i]$   
       $s[j] \leftarrow i$   
return  $r[n]$ 
```

Algorithm BOTTOM-UP-CUT-POLE(p, n)

- $s[i]$ contains position of first cut in optimal solution
- Easy to reconstruct all cuts

Subproblem Graph

Subproblem Graph

- One node for each subproblem

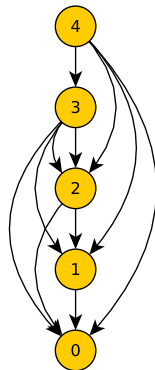
Subproblem Graph

- One node for each subproblem
- Directed edge from a subproblem A to subproblem B if the solution of A depends on the solution of B

Subproblem Graph

- One node for each subproblem
- Directed edge from a subproblem A to subproblem B if the solution of A depends on the solution of B

Example: POLE-CUTTING

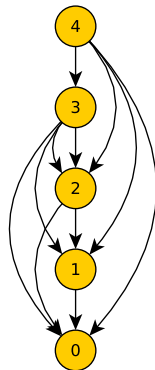


Subproblem Graph

- One node for each subproblem
- Directed edge from a subproblem A to subproblem B if the solution of A depends on the solution of B

Example: POLE-CUTTING

Runtime of Dynamic Programming Algorithm:



Subproblem Graph and Runtime

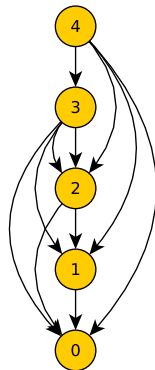
Subproblem Graph

- One node for each subproblem
- Directed edge from a subproblem A to subproblem B if the solution of A depends on the solution of B

Example: POLE-CUTTING

Runtime of Dynamic Programming Algorithm:

- Total number of subproblems t



Subproblem Graph and Runtime

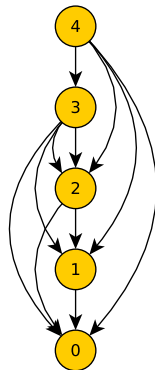
Subproblem Graph

- One node for each subproblem
- Directed edge from a subproblem A to subproblem B if the solution of A depends on the solution of B

Example: POLE-CUTTING

Runtime of Dynamic Programming Algorithm:

- Total number of subproblems t
- Maximum number of subproblems a subproblem depends on s



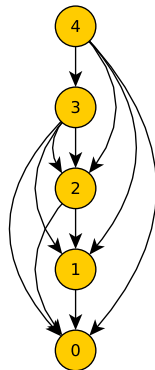
Subproblem Graph

- One node for each subproblem
- Directed edge from a subproblem A to subproblem B if the solution of A depends on the solution of B

Example: POLE-CUTTING

Runtime of Dynamic Programming Algorithm:

- Total number of subproblems t
- Maximum number of subproblems a subproblem depends on s
- Runtime: $O(s \cdot t)$ (assuming that computing solution takes time $O(s)$)



Fibonacci Numbers

Fibonacci Numbers:

$$F_0 = 0, F_1 = 1$$

$$F_n = F_{n-1} + F_{n-2} \text{ for } n \geq 2 .$$

Require: Integer $n \geq 0$

if $n \leq 1$ **then**

return n

else

$A \leftarrow$ array of size n

$A[0] \leftarrow 1, A[1] \leftarrow 1$

for $i \leftarrow 2 \dots n$ **do**

$A[i] \leftarrow A[i-2] + A[i-1]$

return $A[n]$

DYNPRGFIB(n)

Why is this a dynamic programming algorithm?

Identify Optimal Substructure:

Identify Optimal Substructure:

- Recall: $F_n = F_{n-1} + F_{n-2}$

Identify Optimal Substructure:

- Recall: $F_n = F_{n-1} + F_{n-2}$
- (Optimal) solution to size n problem equals sum of (optimal) solutions to subproblems of sizes $n - 1$ and $n - 2$ ✓

Identify Optimal Substructure:

- Recall: $F_n = F_{n-1} + F_{n-2}$
- (Optimal) solution to size n problem equals sum of (optimal) solutions to subproblems of sizes $n - 1$ and $n - 2$ ✓

Give Recursive Solution:

Identify Optimal Substructure:

- Recall: $F_n = F_{n-1} + F_{n-2}$
- (Optimal) solution to size n problem equals sum of (optimal) solutions to subproblems of sizes $n - 1$ and $n - 2$ ✓

Give Recursive Solution:

- Recursive solution is already given in the problem description

Identify Optimal Substructure:

- Recall: $F_n = F_{n-1} + F_{n-2}$
- (Optimal) solution to size n problem equals sum of (optimal) solutions to subproblems of sizes $n - 1$ and $n - 2$ ✓

Give Recursive Solution:

- Recursive solution is already given in the problem description
- $F_n = F_{n-1} + F_{n-2}$

Identify Optimal Substructure:

- Recall: $F_n = F_{n-1} + F_{n-2}$
- (Optimal) solution to size n problem equals sum of (optimal) solutions to subproblems of sizes $n - 1$ and $n - 2$ ✓

Give Recursive Solution:

- Recursive solution is already given in the problem description
- $F_n = F_{n-1} + F_{n-2}$

Compute Optimal Costs & Compute Optimal Solution

Identify Optimal Substructure:

- Recall: $F_n = F_{n-1} + F_{n-2}$
- (Optimal) solution to size n problem equals sum of (optimal) solutions to subproblems of sizes $n - 1$ and $n - 2$ ✓

Give Recursive Solution:

- Recursive solution is already given in the problem description
- $F_n = F_{n-1} + F_{n-2}$

Compute Optimal Costs & Compute Optimal Solution

- Cost and solution is identical for Fibonacci numbers

Identify Optimal Substructure:

- Recall: $F_n = F_{n-1} + F_{n-2}$
- (Optimal) solution to size n problem equals sum of (optimal) solutions to subproblems of sizes $n - 1$ and $n - 2$ ✓

Give Recursive Solution:

- Recursive solution is already given in the problem description
- $F_n = F_{n-1} + F_{n-2}$

Compute Optimal Costs & Compute Optimal Solution

- Cost and solution is identical for Fibonacci numbers
- There is no need to keep track of optimal choices, since there is only a single choice

Maximum Subarray Problem

Problem: MAXIMUM-SUBARRAY

Maximum Subarray Problem

Problem: MAXIMUM-SUBARRAY

- **Input:** Array A of n numbers
- **Output:** Indices $0 \leq i \leq j \leq n - 1$ such that $\sum_{l=i}^j A[l]$ is maximum.

Maximum Subarray Problem

Problem: MAXIMUM-SUBARRAY

- **Input:** Array A of n numbers
- **Output:** Indices $0 \leq i \leq j \leq n - 1$ such that $\sum_{l=i}^j A[l]$ is maximum.

Example:

Maximum Subarray Problem

Problem: MAXIMUM-SUBARRAY

- **Input:** Array A of n numbers
- **Output:** Indices $0 \leq i \leq j \leq n - 1$ such that $\sum_{l=i}^j A[l]$ is maximum.

Example:

-25 20 -3 -16 -23 18 20 -7 12 -5 1

Maximum Subarray Problem

Problem: MAXIMUM-SUBARRAY

- **Input:** Array A of n numbers
- **Output:** Indices $0 \leq i \leq j \leq n - 1$ such that $\sum_{l=i}^j A[l]$ is maximum.

Example:

-25 20 -3 -16 -23 18 20 -7 12 -5 1

Maximum Subarray Problem

Problem: MAXIMUM-SUBARRAY

- **Input:** Array A of n numbers
- **Output:** Indices $0 \leq i \leq j \leq n - 1$ such that $\sum_{l=i}^j A[l]$ is maximum.

Example:

-25 20 -3 -16 -23 18 20 -7 12 -5 1

Divide-and-Conquer Algorithm

- In lecture 7 we gave a divide-and-conquer algorithm with runtime $O(n \log n)$

Maximum Subarray Problem

Problem: MAXIMUM-SUBARRAY

- **Input:** Array A of n numbers
- **Output:** Indices $0 \leq i \leq j \leq n - 1$ such that $\sum_{l=i}^j A[l]$ is maximum.

Example:

-25 20 -3 -16 -23 18 20 -7 12 -5 1

Divide-and-Conquer Algorithm

- In lecture 7 we gave a divide-and-conquer algorithm with runtime $O(n \log n)$
- We will give now a faster dynamic programming algorithm

Related Problem: MAXIMUM-SUFFIX-ARRAY

Related Problem: MAXIMUM-SUFFIX-ARRAY

- **Input:** Array A of n numbers
- **Output:** Index $0 \leq i \leq n - 1$ such that $\sum_{l=i}^{n-1} A[l]$ is maximum.

Related Problem: MAXIMUM-SUFFIX-ARRAY

- **Input:** Array A of n numbers
- **Output:** Index $0 \leq i \leq n - 1$ such that $\sum_{l=i}^{n-1} A[l]$ is maximum.

-25 20 -3 -16 -23 18 20 -7 12 -5 1

Related Problem: MAXIMUM-SUFFIX-ARRAY

- **Input:** Array A of n numbers
- **Output:** Index $0 \leq i \leq n - 1$ such that $\sum_{l=i}^{n-1} A[l]$ is maximum.

-25 20 -3 -16 -23 18 20 -7 12 -5 1

Related Problem: MAXIMUM-SUFFIX-ARRAY

- **Input:** Array A of n numbers
- **Output:** Index $0 \leq i \leq n - 1$ such that $\sum_{l=i}^{n-1} A[l]$ is maximum.

-25 20 -3 -16 -23 18 20 -7 12 -5 1

Optimal Substructure for MAXIMUM-SUBARRAY:

Related Problem: MAXIMUM-SUFFIX-ARRAY

- **Input:** Array A of n numbers
- **Output:** Index $0 \leq i \leq n - 1$ such that $\sum_{l=i}^{n-1} A[l]$ is maximum.

-25 20 -3 -16 -23 18 20 -7 12 -5 1

Optimal Substructure for MAXIMUM-SUBARRAY:

- Let i, j be the indices of the optimal solution

Related Problem: MAXIMUM-SUFFIX-ARRAY

- **Input:** Array A of n numbers
- **Output:** Index $0 \leq i \leq n - 1$ such that $\sum_{l=i}^{n-1} A[l]$ is maximum.

-25 20 -3 -16 -23 18 20 -7 12 -5 1

Optimal Substructure for MAXIMUM-SUBARRAY:

- Let i, j be the indices of the optimal solution
- Then i is the optimal solution for MAXIMUM-SUFFIX-ARRAY on input $A[0 \dots j]$

Related Problem: MAXIMUM-SUFFIX-ARRAY

- **Input:** Array A of n numbers
- **Output:** Index $0 \leq i \leq n - 1$ such that $\sum_{l=i}^{n-1} A[l]$ is maximum.

-25 20 -3 -16 -23 18 20 -7 12 -5 1

Optimal Substructure for MAXIMUM-SUBARRAY:

- Let i, j be the indices of the optimal solution
- Then i is the optimal solution for MAXIMUM-SUFFIX-ARRAY on input $A[0 \dots j]$

-25 20 -3 -16 -23 18 20 -7 12 -5 1

Related Problem: MAXIMUM-SUFFIX-ARRAY

- **Input:** Array A of n numbers
- **Output:** Index $0 \leq i \leq n - 1$ such that $\sum_{l=i}^{n-1} A[l]$ is maximum.

-25 20 -3 -16 -23 18 20 -7 12 -5 1

Optimal Substructure for MAXIMUM-SUBARRAY:

- Let i, j be the indices of the optimal solution
- Then i is the optimal solution for MAXIMUM-SUFFIX-ARRAY on input $A[0 \dots j]$

-25 20 -3 -16 -23 18 20 -7 12

Dynamic Programming for Maximum Suffix Array

Optimal Substructure:

Optimal Substructure:

Lemma

Let A be an array of length n . Let i be the optimal solution for MAXIMUM-SUFFIX-ARRAY on A . If $i < n - 1$ then the optimal solution to MAXIMUM-SUFFIX-ARRAY on $A[0 \dots n - 2]$ is also i .

Optimal Substructure:

Lemma

Let A be an array of length n . Let i be the optimal solution for MAXIMUM-SUFFIX-ARRAY on A . If $i < n - 1$ then the optimal solution to MAXIMUM-SUFFIX-ARRAY on $A[0 \dots n - 2]$ is also i .

$A[0]$ $A[1]$ \dots $A[i]$ $A[i + 1]$ \dots $A[n - 2]$ $A[n - 1]$

Optimal Substructure:

Lemma

Let A be an array of length n . Let i be the optimal solution for MAXIMUM-SUFFIX-ARRAY on A . If $i < n - 1$ then the optimal solution to MAXIMUM-SUFFIX-ARRAY on $A[0 \dots n - 2]$ is also i .

$A[0]$ $A[1]$ \dots $A[i]$ $A[i + 1]$ \dots $A[n - 2]$ $A[n - 1]$

Proof.

Optimal Substructure:

Lemma

Let A be an array of length n . Let i be the optimal solution for MAXIMUM-SUFFIX-ARRAY on A . If $i < n - 1$ then the optimal solution to MAXIMUM-SUFFIX-ARRAY on $A[0 \dots n - 2]$ is also i .

$A[0]$ $A[1]$ \dots $A[i]$ $A[i + 1]$ \dots $A[n - 2]$ $A[n - 1]$

Proof. Suppose that the lemma is not true and suppose that $i' \neq i$ is the optimal solution to MAXIMUM-SUFFIX-ARRAY on $A[0 \dots n - 2]$.

Optimal Substructure:

Lemma

Let A be an array of length n . Let i be the optimal solution for MAXIMUM-SUFFIX-ARRAY on A . If $i < n - 1$ then the optimal solution to MAXIMUM-SUFFIX-ARRAY on $A[0 \dots n - 2]$ is also i .

$A[0]$ $A[1]$... $A[i]$ $A[i + 1]$... $A[n - 2]$ $A[n - 1]$

Proof. Suppose that the lemma is not true and suppose that $i' \neq i$ is the optimal solution to MAXIMUM-SUFFIX-ARRAY on $A[0 \dots n - 2]$. Then,

$$\sum_{j=i'}^{n-2} A[j] > \sum_{j=i}^{n-2} A[j]$$

Optimal Substructure:

Lemma

Let A be an array of length n . Let i be the optimal solution for MAXIMUM-SUFFIX-ARRAY on A . If $i < n - 1$ then the optimal solution to MAXIMUM-SUFFIX-ARRAY on $A[0 \dots n - 2]$ is also i .

$$A[0] \quad A[1] \quad \dots \quad A[i] \quad A[i + 1] \quad \dots \quad A[n - 2] \quad A[n - 1]$$

Proof. Suppose that the lemma is not true and suppose that $i' \neq i$ is the optimal solution to MAXIMUM-SUFFIX-ARRAY on $A[0 \dots n - 2]$. Then,

$$\sum_{j=i'}^{n-2} A[j] > \sum_{j=i}^{n-2} A[j]$$

But then $\sum_{j=i'}^{n-1} A[j] > \sum_{j=i}^{n-1} A[j]$,

Optimal Substructure:

Lemma

Let A be an array of length n . Let i be the optimal solution for MAXIMUM-SUFFIX-ARRAY on A . If $i < n - 1$ then the optimal solution to MAXIMUM-SUFFIX-ARRAY on $A[0 \dots n - 2]$ is also i .

$A[0]$ $A[1]$ \dots $A[i]$ $A[i + 1]$ \dots $A[n - 2]$ $A[n - 1]$

Proof. Suppose that the lemma is not true and suppose that $i' \neq i$ is the optimal solution to MAXIMUM-SUFFIX-ARRAY on $A[0 \dots n - 2]$. Then,

$$\sum_{j=i'}^{n-2} A[j] > \sum_{j=i}^{n-2} A[j]$$

But then $\sum_{j=i'}^{n-1} A[j] > \sum_{j=i}^{n-1} A[j]$, a contradiction to the fact that i is optimal for A . □

Recursive Solution to Maximum Suffix Array

Recursive Solution:

Recursive Solution:

$m[i] :=$ value of maximum suffix array of $A[0 \dots i]$

Recursive Solution:

$m[i] :=$ value of maximum suffix array of $A[0 \dots i]$

$$m[i] = \begin{cases} A[0] & \text{if } i = 0 \\ A[i] & \text{if } m[i-1] \leq 0 \\ m[i-1] + A[i] & \text{if } m[i-1] > 0 . \end{cases}$$

Recursive Solution to Maximum Suffix Array

Recursive Solution:

$m[i] :=$ value of maximum suffix array of $A[0 \dots i]$

$$m[i] = \begin{cases} A[0] & \text{if } i = 0 \\ A[i] & \text{if } m[i - 1] \leq 0 \\ m[i - 1] + A[i] & \text{if } m[i - 1] > 0 . \end{cases}$$

Example: Bottom-up Computation

Recursive Solution to Maximum Suffix Array

Recursive Solution:

$m[i] :=$ value of maximum suffix array of $A[0 \dots i]$

$$m[i] = \begin{cases} A[0] & \text{if } i = 0 \\ A[i] & \text{if } m[i-1] \leq 0 \\ m[i-1] + A[i] & \text{if } m[i-1] > 0. \end{cases}$$

Example: Bottom-up Computation

	0	1	2	3	4	5	6	7	8	9	10
A	-25	20	-3	-16	-23	18	20	-7	12	-5	1
m											

Recursive Solution to Maximum Suffix Array

Recursive Solution:

$m[i] :=$ value of maximum suffix array of $A[0 \dots i]$

$$m[i] = \begin{cases} A[0] & \text{if } i = 0 \\ A[i] & \text{if } m[i-1] \leq 0 \\ m[i-1] + A[i] & \text{if } m[i-1] > 0. \end{cases}$$

Example: Bottom-up Computation

	0	1	2	3	4	5	6	7	8	9	10
A	-25	20	-3	-16	-23	18	20	-7	12	-5	1
m	-25										

Recursive Solution to Maximum Suffix Array

Recursive Solution:

$m[i] :=$ value of maximum suffix array of $A[0 \dots i]$

$$m[i] = \begin{cases} A[0] & \text{if } i = 0 \\ A[i] & \text{if } m[i-1] \leq 0 \\ m[i-1] + A[i] & \text{if } m[i-1] > 0. \end{cases}$$

Example: Bottom-up Computation

	0	1	2	3	4	5	6	7	8	9	10
A	-25	20	-3	-16	-23	18	20	-7	12	-5	1
m	-25	20									

Recursive Solution to Maximum Suffix Array

Recursive Solution:

$m[i] :=$ value of maximum suffix array of $A[0 \dots i]$

$$m[i] = \begin{cases} A[0] & \text{if } i = 0 \\ A[i] & \text{if } m[i-1] \leq 0 \\ m[i-1] + A[i] & \text{if } m[i-1] > 0. \end{cases}$$

Example: Bottom-up Computation

	0	1	2	3	4	5	6	7	8	9	10
A	-25	20	-3	-16	-23	18	20	-7	12	-5	1
m	-25	20	17								

Recursive Solution to Maximum Suffix Array

Recursive Solution:

$m[i] :=$ value of maximum suffix array of $A[0 \dots i]$

$$m[i] = \begin{cases} A[0] & \text{if } i = 0 \\ A[i] & \text{if } m[i-1] \leq 0 \\ m[i-1] + A[i] & \text{if } m[i-1] > 0. \end{cases}$$

Example: Bottom-up Computation

	0	1	2	3	4	5	6	7	8	9	10
A	-25	20	-3	-16	-23	18	20	-7	12	-5	1
m	-25	20	17	1							

Recursive Solution to Maximum Suffix Array

Recursive Solution:

$m[i] :=$ value of maximum suffix array of $A[0 \dots i]$

$$m[i] = \begin{cases} A[0] & \text{if } i = 0 \\ A[i] & \text{if } m[i-1] \leq 0 \\ m[i-1] + A[i] & \text{if } m[i-1] > 0. \end{cases}$$

Example: Bottom-up Computation

	0	1	2	3	4	5	6	7	8	9	10
A	-25	20	-3	-16	-23	18	20	-7	12	-5	1
m	-25	20	17	1	-22						

Recursive Solution to Maximum Suffix Array

Recursive Solution:

$m[i] :=$ value of maximum suffix array of $A[0 \dots i]$

$$m[i] = \begin{cases} A[0] & \text{if } i = 0 \\ A[i] & \text{if } m[i-1] \leq 0 \\ m[i-1] + A[i] & \text{if } m[i-1] > 0. \end{cases}$$

Example: Bottom-up Computation

	0	1	2	3	4	5	6	7	8	9	10
<i>A</i>	-25	20	-3	-16	-23	18	20	-7	12	-5	1
<i>m</i>	-25	20	17	1	-22	18					

Recursive Solution to Maximum Suffix Array

Recursive Solution:

$m[i] :=$ value of maximum suffix array of $A[0 \dots i]$

$$m[i] = \begin{cases} A[0] & \text{if } i = 0 \\ A[i] & \text{if } m[i-1] \leq 0 \\ m[i-1] + A[i] & \text{if } m[i-1] > 0. \end{cases}$$

Example: Bottom-up Computation

	0	1	2	3	4	5	6	7	8	9	10
<i>A</i>	-25	20	-3	-16	-23	18	20	-7	12	-5	1
<i>m</i>	-25	20	17	1	-22	18	38				

Recursive Solution to Maximum Suffix Array

Recursive Solution:

$m[i] :=$ value of maximum suffix array of $A[0 \dots i]$

$$m[i] = \begin{cases} A[0] & \text{if } i = 0 \\ A[i] & \text{if } m[i-1] \leq 0 \\ m[i-1] + A[i] & \text{if } m[i-1] > 0. \end{cases}$$

Example: Bottom-up Computation

	0	1	2	3	4	5	6	7	8	9	10
<i>A</i>	-25	20	-3	-16	-23	18	20	-7	12	-5	1
<i>m</i>	-25	20	17	1	-22	18	38	31			

Recursive Solution to Maximum Suffix Array

Recursive Solution:

$m[i] :=$ value of maximum suffix array of $A[0 \dots i]$

$$m[i] = \begin{cases} A[0] & \text{if } i = 0 \\ A[i] & \text{if } m[i-1] \leq 0 \\ m[i-1] + A[i] & \text{if } m[i-1] > 0. \end{cases}$$

Example: Bottom-up Computation

	0	1	2	3	4	5	6	7	8	9	10
<i>A</i>	-25	20	-3	-16	-23	18	20	-7	12	-5	1
<i>m</i>	-25	20	17	1	-22	18	38	31	43		

Recursive Solution to Maximum Suffix Array

Recursive Solution:

$m[i] :=$ value of maximum suffix array of $A[0 \dots i]$

$$m[i] = \begin{cases} A[0] & \text{if } i = 0 \\ A[i] & \text{if } m[i-1] \leq 0 \\ m[i-1] + A[i] & \text{if } m[i-1] > 0. \end{cases}$$

Example: Bottom-up Computation

	0	1	2	3	4	5	6	7	8	9	10
<i>A</i>	-25	20	-3	-16	-23	18	20	-7	12	-5	1
<i>m</i>	-25	20	17	1	-22	18	38	31	43	38	

Recursive Solution to Maximum Suffix Array

Recursive Solution:

$m[i] :=$ value of maximum suffix array of $A[0 \dots i]$

$$m[i] = \begin{cases} A[0] & \text{if } i = 0 \\ A[i] & \text{if } m[i-1] \leq 0 \\ m[i-1] + A[i] & \text{if } m[i-1] > 0. \end{cases}$$

Example: Bottom-up Computation

	0	1	2	3	4	5	6	7	8	9	10
<i>A</i>	-25	20	-3	-16	-23	18	20	-7	12	-5	1
<i>m</i>	-25	20	17	1	-22	18	38	31	43	38	39

Recursive Solution to Maximum Suffix Array

Recursive Solution:

$m[i] :=$ value of maximum suffix array of $A[0 \dots i]$

$$m[i] = \begin{cases} A[0] & \text{if } i = 0 \\ A[i] & \text{if } m[i-1] \leq 0 \\ m[i-1] + A[i] & \text{if } m[i-1] > 0. \end{cases}$$

Example: Bottom-up Computation

	0	1	2	3	4	5	6	7	8	9	10
A	-25	20	-3	-16	-23	18	20	-7	12	-5	1
m	-25	20	17	1	-22	18	38	31	43	38	39

Maximum constitutes optimal solution to MAXIMUM-SUBARRAY!

Dynamic Programming Algorithm for Maximum Subarray

Algorithm: Input is an array A of integers of length n

- 1 Compute dyn. prog. table for MAXIMUM-SUFFIX-ARRAY
- 2 Return the maximum value in the table

Require: Array A of n integers

Let $m[0 \dots n - 1]$ be a new array

$m[0] \leftarrow A[0]$

$q \leftarrow A[0]$

for $i = 1 \dots n - 1$ **do**

if $m[i - 1] < 0$ **then**

$m[i] \leftarrow A[i]$

else

$m[i] \leftarrow A[i] + m[i - 1]$

$q \leftarrow \max\{q, m[i]\}$

return q

Kadane's Algorithm for MAXIMUM-SUBARRAY

Kadane's Algorithm



Kadane's Algorithm

- Runtime: $O(n)$ (n subproblems, only one subproblem needed to compute current value)



Kadane's Algorithm

- Runtime: $O(n)$ (n subproblems, only one subproblem needed to compute current value)
- Recall that Divide-and-Conquer solution has a runtime of $O(n \log n)$



Kadane's Algorithm

- Runtime: $O(n)$ (n subproblems, only one subproblem needed to compute current value)
- Recall that Divide-and-Conquer solution has a runtime of $O(n \log n)$
- Observe that for MAXIMUM-SUBARRAY Dynamic Programming and Divide-and-Conquer is applicable

Challenges:

- Compute max. subarray of size at most k , for some k
- Compute subarray $A[i, j]$ such that

$$\frac{\sum_{k=i}^j A[k]}{\sqrt{j-i+1}}$$

is maximized.

