



CPSC 490 – Problem Solving in Computer Science

Lecture 5: Dynamic Programming (Part 1)

Jason Chiu and Raunak Kumar

2017/01/13

University of British Columbia

What is Dynamic Programming

DP algorithm = recursive function

What is Dynamic Programming

DP algorithm = recursive function

Solving a problem with DP involves these three steps:

1. Characterize problem and subproblems (i.e. find the DP state)
 - Suppose we represent the answer to the problem as $f(n)$...
 - Then, can we use $f(n - 1), f(n - 2), \dots$ to quickly compute $f(n)$?

What is Dynamic Programming

DP algorithm = recursive function

Solving a problem with DP involves these three steps:

1. Characterize problem and subproblems (i.e. find the DP state)
 - Suppose we represent the answer to the problem as $f(n)$...
 - Then, can we use $f(n - 1), f(n - 2), \dots$ to quickly compute $f(n)$?
2. Find the recurrence relation
 - $f(n) = \text{some function of } f(n - 1), f(n - 2), \dots$
 - $f(1) = \dots$
 - **NO CYCLES!**

What is Dynamic Programming

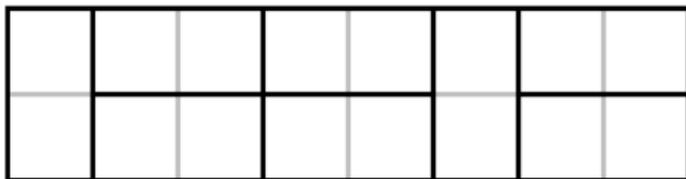
DP algorithm = recursive function

Solving a problem with DP involves these three steps:

1. Characterize problem and subproblems (i.e. find the DP state)
 - Suppose we represent the answer to the problem as $f(n)$...
 - Then, can we use $f(n - 1), f(n - 2), \dots$ to quickly compute $f(n)$?
2. Find the recurrence relation
 - $f(n) = \text{some function of } f(n - 1), f(n - 2), \dots$
 - $f(1) = \dots$
 - **NO CYCLES!**
3. Optimize by reusing previously computed values
 - Memoization: implement the function using recursion, but do not recompute $f(k)$ if we have already done it before
 - Bottom-up DP: instead of recursion, compute $f(1), \dots, f(n)$ in order

Example 1

How many ways are there to tile a $2 \times n$ grid with 2×1 tiles?



Example 1 – Solution

DP state: $f(k)$ = number of ways to tile $2 \times k$ grid with 2×1 tiles.

Example 1 – Solution

DP state: $f(k)$ = number of ways to tile $2 \times k$ grid with 2×1 tiles.

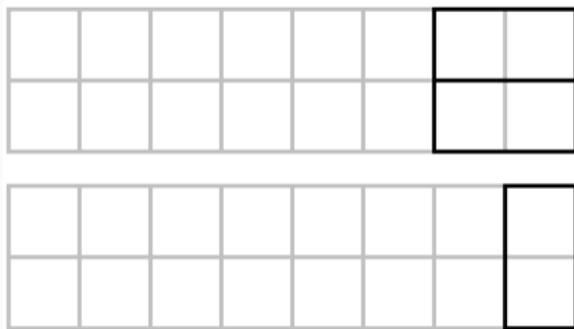
Base case: $f(0) = f(1) = 1$.

Example 1 – Solution

DP state: $f(k)$ = number of ways to tile $2 \times k$ grid with 2×1 tiles.

Base case: $f(0) = f(1) = 1$.

To compute $f(k)$ we consider the “last” tiles we use to tile the $2 \times k$ grid and we see that there are only these two cases:

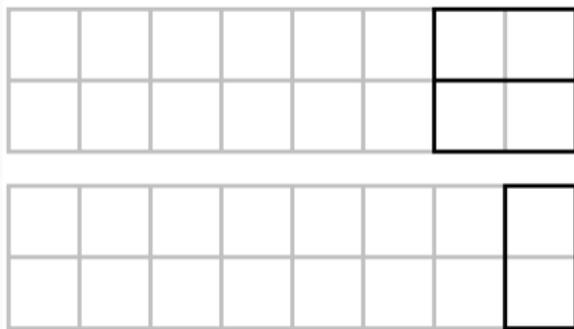


Example 1 – Solution

DP state: $f(k)$ = number of ways to tile $2 \times k$ grid with 2×1 tiles.

Base case: $f(0) = f(1) = 1$.

To compute $f(k)$ we consider the “last” tiles we use to tile the $2 \times k$ grid and we see that there are only these two cases:



Recurrence: $f(k) = f(k - 1) + f(k - 2)$

Answer: $f(n)$

Time complexity: $O(n)$

Example 1 – Implementation

Memoization

```
1  int memo[N]; // memset to -1
2  int f(int n) {
3      if (n == 0 || n == 1) return 1;
4      if (memo[n] >= 0) return memo[n];
5      return memo[n] = f(n-1) + f(n-2);
6  }
```

Bottom-up Dynamic Programming

```
1  int f[N];
2  f[0] = f[1] = 1;
3  for (int i = 2; i < N; i++) {
4      f[i] = f[i-1] + f[i-2];
5  }
```

Example 2 – Longest Increasing Subsequence

Find the length of the longest increasing subsequence (LIS) in an array of n integers.

[10, 11, 1, 5, 3, 9, -1, 25]

Example 2 – LIS Solution 1

DP state: $f(k)$ = length of the LIS of $A[1 \dots k]$ ending at $A[k]$.

Example 2 – LIS Solution 1

DP state: $f(k)$ = length of the LIS of $A[1 \dots k]$ ending at $A[k]$.

Base case: $f(0) = 1$.

Example 2 – LIS Solution 1

DP state: $f(k)$ = length of the LIS of $A[1 \dots k]$ ending at $A[k]$.

Base case: $f(0) = 1$.

To get LIS ending at the $A[k]$ we take the element $A[k]$ and append it to a LIS ending at a previous element $A[i]$ that is smaller.

[10, 11, 1, 5, 3, 9, -1, 25]

Example 2 – LIS Solution 1

DP state: $f(k)$ = length of the LIS of $A[1 \dots k]$ ending at $A[k]$.

Base case: $f(0) = 1$.

To get LIS ending at the $A[k]$ we take the element $A[k]$ and append it to a LIS ending at a previous element $A[i]$ that is smaller.

[10, 11, 1, 5, 3, 9, -1, 25]

Example 2 – LIS Solution 1

DP state: $f(k)$ = length of the LIS of $A[1 \dots k]$ ending at $A[k]$.

Base case: $f(0) = 1$.

To get LIS ending at the $A[k]$ we take the element $A[k]$ and append it to a LIS ending at a previous element $A[i]$ that is smaller.

[10, 11, 1, 5, 3, 9, -1, 25]

Example 2 – LIS Solution 1

DP state: $f(k)$ = length of the LIS of $A[1 \dots k]$ ending at $A[k]$.

Base case: $f(0) = 1$.

To get LIS ending at the $A[k]$ we take the element $A[k]$ and append it to a LIS ending at a previous element $A[i]$ that is smaller.

[10, 11, 1, 5, 3, 9, -1, 25]

Example 2 – LIS Solution 1

DP state: $f(k)$ = length of the LIS of $A[1 \dots k]$ ending at $A[k]$.

Base case: $f(0) = 1$.

To get LIS ending at the $A[k]$ we take the element $A[k]$ and append it to a LIS ending at a previous element $A[i]$ that is smaller.

[10, 11, 1, 5, 3, 9, -1, 25]

Example 2 – LIS Solution 1

DP state: $f(k)$ = length of the LIS of $A[1 \dots k]$ ending at $A[k]$.

Base case: $f(0) = 1$.

To get LIS ending at the $A[k]$ we take the element $A[k]$ and append it to a LIS ending at a previous element $A[i]$ that is smaller.

[10, 11, 1, 5, 3, 9, -1, 25]

Example 2 – LIS Solution 1

DP state: $f(k)$ = length of the LIS of $A[1 \dots k]$ ending at $A[k]$.

Base case: $f(0) = 1$.

To get LIS ending at the $A[k]$ we take the element $A[k]$ and append it to a LIS ending at a previous element $A[i]$ that is smaller.

[10, 11, 1, 5, 3, 9, -1, 25]

Example 2 – LIS Solution 1

DP state: $f(k)$ = length of the LIS of $A[1 \dots k]$ ending at $A[k]$.

Base case: $f(0) = 1$.

To get LIS ending at the $A[k]$ we take the element $A[k]$ and append it to a LIS ending at a previous element $A[i]$ that is smaller.

Recurrence: $f(k) = 1 + \max f(i)$ where $0 \leq i < k$ and $A[i] < A[k]$

Answer: $\max f(k)$ where $0 \leq k \leq n$

Time complexity: $O(n^2)$

Example 2 – LIS Solution 1

DP state: $f(k)$ = length of the LIS of $A[1 \dots k]$ ending at $A[k]$.

Base case: $f(0) = 1$.

To get LIS ending at the $A[k]$ we take the element $A[k]$ and append it to a LIS ending at a previous element $A[i]$ that is smaller.

Recurrence: $f(k) = 1 + \max f(i)$ where $0 \leq i < k$ and $A[i] < A[k]$

Answer: $\max f(k)$ where $0 \leq k \leq n$

Time complexity: $O(n^2)$

Can we do better?

Technique: Flip the DP State

We had $f(k) = i$, why not do $L(i) = k$ instead??

Instead of $f(k) = \text{length of LIS ending at index } k$

we do $L(i) = \text{smallest element ending a LIS of length } i$

Example 2 – LIS Solution 2

DP state: $L(i)$ = smallest element ending LIS of length i , or ∞

Example 2 – LIS Solution 2

DP state: $L(i)$ = smallest element ending LIS of length i , or ∞

Base case: initialize $L(i)$ to ∞ for all i

Example 2 – LIS Solution 2

DP state: $L(i)$ = smallest element ending LIS of length i , or ∞

Base case: initialize $L(i)$ to ∞ for all i

Suppose our dp state L is valid for the subarray $A[0..k-1]$, and we encounter a new element $A[k]$, how do we update L ?

Example 2 – LIS Solution 2

DP state: $L(i)$ = smallest element ending LIS of length i , or ∞

Base case: initialize $L(i)$ to ∞ for all i

Suppose our dp state L is valid for the subarray $A[0..k-1]$, and we encounter a new element $A[k]$, how do we update L ?

Observation 1: L is monotonically increasing

Example 2 – LIS Solution 2

DP state: $L(i)$ = smallest element ending LIS of length i , or ∞

Base case: initialize $L(i)$ to ∞ for all i

Suppose our dp state L is valid for the subarray $A[0..k-1]$, and we encounter a new element $A[k]$, how do we update L ?

Observation 1: L is monotonically increasing

Observation 2: only need to update the first element $> A[k]$ to $A[k]$

Example 2 – LIS Solution 2

DP state: $L(i)$ = smallest element ending LIS of length i , or ∞

Base case: initialize $L(i)$ to ∞ for all i

Suppose our dp state L is valid for the subarray $A[0..k-1]$, and we encounter a new element $A[k]$, how do we update L ?

Observation 1: L is monotonically increasing

Observation 2: only need to update the first element $> A[k]$ to $A[k]$

We can binary search to find this element to update!

Example 2 – LIS Solution 2

DP state: $L(i)$ = smallest element ending LIS of length i , or ∞

Base case: initialize $L(i)$ to ∞ for all i

Suppose our dp state L is valid for the subarray $A[0..k-1]$, and we encounter a new element $A[k]$, how do we update L ?

Observation 1: L is monotonically increasing

Observation 2: only need to update the first element $> A[k]$ to $A[k]$

We can binary search to find this element to update!

Answer: the biggest i such that $L(i) < \infty$

Time complexity: $O(n \log n)$

Example 3 – Knapsack

You have n items, each with weight w_i and value v_i

You can carry maximum weight W

Maximize total value you can carry, taking at most one of each item.

Value	1	10	3	7
Weight	1	9	2	3

If $W = 9$ then maximum total value = 11

Example 3 – Solution

DP state: $f(n, k)$ = Maximum value of items that can be placed in the knapsack from items $\{1, 2, \dots, n\}$ given capacity k .

Example 3 – Solution

DP state: $f(n, k)$ = Maximum value of items that can be placed in the knapsack from items $\{1, 2, \dots, n\}$ given capacity k .

Base case: $f(0, k) = 0$.

Example 3 – Solution

DP state: $f(n, k)$ = Maximum value of items that can be placed in the knapsack from items $\{1, 2, \dots, n\}$ given capacity k .

Base case: $f(0, k) = 0$.

To compute $f(n, k)$, consider two cases: either add the n^{th} item to our knapsack or don't add it.

Example 3 – Solution

DP state: $f(n, k)$ = Maximum value of items that can be placed in the knapsack from items $\{1, 2, \dots, n\}$ given capacity k .

Base case: $f(0, k) = 0$.

To compute $f(n, k)$, consider two cases: either add the n^{th} item to our knapsack or don't add it.

Recurrence relation:

$$f(n, k) = \begin{cases} f(n-1, k) & \text{if } w_n > k \\ \max(f(n-1, k), f(n-1, k - w_n) + v_n) & \text{otherwise} \end{cases}$$

Answer: $f(n, W)$

Time complexity: $O(nW)$

What if we can use each item as many times as we want?

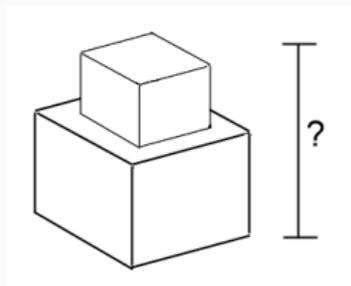
Problem 1

You are given a set of n 3-dimensional boxes.

You can place box i on top of box j if the dimensions of the base of box i are strictly smaller than that of the base of box j .

You can rotate the boxes and use multiple copies of the same box.

What is the height of the tallest stack that can be made?



Problem 1 – Solution

We can rotate box and use multiple copies of same box

Problem 1 – Solution

We can rotate box and use multiple copies of same box
→ Create 3 copies of each box (1 for each rotation)

Problem 1 – Solution

We can rotate box and use multiple copies of same box

→ Create 3 copies of each box (1 for each rotation)

Box i on top of box j only if both dimensions of the base are smaller.

Problem 1 – Solution

We can rotate box and use multiple copies of same box

→ Create 3 copies of each box (1 for each rotation)

Box i on top of box j only if both dimensions of the base are smaller.

→ Base area of box $i <$ base area of box j

Problem 1 – Solution

We can rotate box and use multiple copies of same box

→ Create 3 copies of each box (1 for each rotation)

Box i on top of box j only if both dimensions of the base are smaller.

→ Base area of box i < base area of box j

→ Sort boxes in desc. order of base area

Problem 1 – Solution (Continued)

DP state: $f(i)$ = Height of the tallest stack from boxes $1, 2, \dots, i$ with box i on top.

Problem 1 – Solution (Continued)

DP state: $f(i)$ = Height of the tallest stack from boxes $1, 2, \dots, i$ with box i on top.

Base case: $f(1) = \text{Box}[1].\text{height}$.

Problem 1 – Solution (Continued)

DP state: $f(i)$ = Height of the tallest stack from boxes $1, 2, \dots, i$ with box i on top.

Base case: $f(1) = \text{Box}[1].\text{height}$.

To compute $f(i)$ we consider adding box i onto the best possible stack from $1, 2, \dots, i - 1$.

Problem 1 – Solution (Continued)

DP state: $f(i)$ = Height of the tallest stack from boxes $1, 2, \dots, i$ with box i on top.

Base case: $f(1) = \text{Box}[1].\text{height}$.

To compute $f(i)$ we consider adding box i onto the best possible stack from $1, 2, \dots, i - 1$.

Recurrence relation:

$f(i) = \text{Box}[i].\text{height} + \max f(j)$ where $1 \leq j < i$, can put Box i on Box j

Answer: $\max f(i)$ for $1 \leq i \leq 3n$

Time complexity: $O(n^2)$

Problem 2

You have $m \times n$ grid of integers. You can only move down or right.
Find the number of palindromic paths from $(1, 1)$ to (m, n)

1	2	4	3	5
2	3	4	3	4
2	2	3	2	1

Problem 2 – Solution

DP state: $f(s_i, s_j, e_i, e_j)$ = Number of palindromic paths that start at (s_i, s_j) and end at (e_i, e_j) .

Problem 2 – Solution

DP state: $f(s_i, s_j, e_i, e_j)$ = Number of palindromic paths that start at (s_i, s_j) and end at (e_i, e_j) .

Base case:

$$f(s_i, s_j, e_i, e_j) = \begin{cases} 0 & \text{if cannot move from } (s_i, s_j) \text{ to } (e_i, e_j) \\ & \text{or } A[s_i][s_j] \neq A[e_i][e_j] \\ 1 & \text{if same/adjacent start/end and } A[s_i][s_j] = A[e_i][e_j] \end{cases}$$

Problem 2 – Solution

DP state: $f(s_i, s_j, e_i, e_j)$ = Number of palindromic paths that start at (s_i, s_j) and end at (e_i, e_j) .

Base case:

$$f(s_i, s_j, e_i, e_j) = \begin{cases} 0 & \text{if cannot move from } (s_i, s_j) \text{ to } (e_i, e_j) \\ & \text{or } A[s_i][s_j] \neq A[e_i][e_j] \\ 1 & \text{if same/adjacent start/end and } A[s_i][s_j] = A[e_i][e_j] \end{cases}$$

To compute $f(s_i, s_j, e_i, e_j)$ we sum over the answers to all possible ways of moving from the two endpoints.

Problem 2 – Solution

DP state: $f(s_i, s_j, e_i, e_j)$ = Number of palindromic paths that start at (s_i, s_j) and end at (e_i, e_j) .

Base case:

$$f(s_i, s_j, e_i, e_j) = \begin{cases} 0 & \text{if cannot move from } (s_i, s_j) \text{ to } (e_i, e_j) \\ & \text{or } A[s_i][s_j] \neq A[e_i][e_j] \\ 1 & \text{if same/adjacent start/end and } A[s_i][s_j] = A[e_i][e_j] \end{cases}$$

To compute $f(s_i, s_j, e_i, e_j)$ we sum over the answers to all possible ways of moving from the two endpoints.

Recurrence relation (assuming $A[s_i][s_j] = A[e_i][e_j]$):

$$\begin{aligned} f(s_i, s_j, e_i, e_j) = & f(s_i + 1, s_j, e_i - 1, e_j) + f(s_i + 1, s_j, e_i, e_j - 1) \\ & + f(s_i, s_j + 1, e_i - 1, e_j) + f(s_i, s_j + 1, e_i, e_j - 1) \end{aligned}$$

Time complexity: $O(m^2n^2)$

Problem 3

Let's play a game!

$n = 2k$ coins on the table with values v_1, v_2, \dots, v_n .

Player 1 and 2 alternate taking one coin from either left or right end.

How much does each player get if each player plays optimally assuming that the other player also plays optimally?



Problem 3 – Solution

DP state: $f(i, j)$ = your maximum winning when coins $i \dots j$ remain and it is your turn, assuming both players play optimally

Problem 3 – Solution

DP state: $f(i, j)$ = your maximum winning when coins $i \dots j$ remain and it is your turn, assuming both players play optimally

Base case: $f(i, i) = v_i, f(i, i + 1) = \max(v_i, v_{i+1})$.

Problem 3 – Solution

DP state: $f(i, j)$ = your maximum winning when coins $i \dots j$ remain and it is your turn, assuming both players play optimally

Base case: $f(i, i) = v_i, f(i, i + 1) = \max(v_i, v_{i+1})$.

Recursive case: consider picking either the i^{th} coin or the j^{th} coin and recursing on the possible choices that the opponent can make.

Problem 3 – Solution

DP state: $f(i, j)$ = your maximum winning when coins $i \dots j$ remain and it is your turn, assuming both players play optimally

Base case: $f(i, i) = v_i, f(i, i + 1) = \max(v_i, v_{i+1})$.

Recursive case: consider picking either the i^{th} coin or the j^{th} coin and recursing on the possible choices that the opponent can make.

Recurrence relation:

$$f(i, j) = \max \left(v_i + \min (f(i+2, j), f(i+1, j-1)), v_j + \min (f(i+1, j-1), f(i, j-2)) \right)$$

Answer: $f(1, n)$.

Time complexity: $O(n^2)$

More Dynamic Programming!