Dynamic Programming [1]

Jie Wang

University of Massachusetts Lowell
Department of Computer Science

[1] I thank Prof. Zachary Kissel of Merrimack College for sharing his lecture notes with me; some of the examples presented here are borrowed from his notes.

What is DP?

DP is a technique for solving a recurrence relation consisting of a lot of repeating subproblems.

Example: Compute the Fibonacci sequence $F(i)$, where

$$F(i) = \begin{cases} 1, & \text{if } i = 1 \text{ or } i = 2, \\ F(i-1) + F(i-2), & \text{if } i > 2. \end{cases}$$

A direct implementation:

```
FIB(n)
1  if n == 1 or n == 2
2        return 1
3  else
4        return FIB(n − 1) + FIB(n − 2)
```

Time Analysis

$$T(n) = \begin{cases} \Theta(1), & \text{if } n \in \{1, 2\}, \\ T(n-1) + T(n-2) + \Theta(1), & \text{otherwise.} \end{cases}$$

$$\begin{aligned} T(n) &\geq 2T(n-2) + 1 \\ &= 2(2T(n-4) + 1) + 1 \\ &= \cdots \\ &= 2^k T(n-2k) + \sum_{i=0}^{k-1} 2^i \\ &= 2^{\frac{n-1}{2}} + \sum_{i=0}^{n-2} 2^i \\ &\in \Theta(2^n). \end{aligned}$$

We Can Do Better

- This exponential blowup is due to significant re-computation of repeating subproblems.
- Note that the i-th Fibonacci number is the sum of the previous two Fibonacci numbers. If we save (memoize) the previous numbers in a table, we can cutdown the running time to $\Theta(n)$.

FIB(n)

```
1  if memo[n] ≠ ⊥
2        return memo[n]
3  else
4        if n ≤ 2
5              v = 1
6        else
7              v = FIB(n − 1) + FIB(n − 2)
8        memo[n] = v
9        return v
```

DP for Optimization

DP is often used for solving optimization problems. To do so, first seek if we can devise a recurrence relation for the problem. We can then use DP to achieve a faster algorithm (polynomial vs exponential) if the following two conditions hold:

1. The number of different subproblems is polynomial (a.k.a. overlapping subproblems).
2. The optimal solution contains within it optimal solutions to the subproblems (a.k.a. optimal substructure).

We can then use memoization (store values of previous subproblems) to achieve a poly-time algorithm.

Tricks to Obtain a Recurrence Relation

- **Formulation**. Interpret and formulate the problem by specifying parameters, where parameters represent the size of the problem (original and subproblems).
- **Localization**. For a given subproblem (specified by parameters), figure out how it is affected by smaller subproblems (also specified by parameters). Express this dependency as a recurrence relation. (Note: this is often the hardest part of DP.)

Rod Cutting

Input: Given a rod of length n inches and a table of prices $p[1 \ldots n]$ listing the value of a rod of length i inches.
Output: The maximum revenue r_n obtainable by cutting up the rod and selling the pieces.

Formulation and Localization

- Formulation: Let $r(n)$ denote the maximum revenue that can be obtained by cutting a rod of length n.
  - There are n subproblems.
  - We want to compute $r(n)$.
- Localization:
  - A rod can be cut into two pieces of length i and length $n - i$.
  -

$$r(n) = \begin{cases} \max_{1 \le i \le n}\{p[i] + r(n-i)\} & \text{If } n > 0, \\ 0 & \text{Otherwise.} \end{cases}$$

Memoization

Use an array $memo[1..n]$ to record results of subproblems.

$\textsc{CutRod}(p[1..n], n)$

```
 1  if memo[n] ≠ ⊥
 2        v = memo[n]
 3  elseif n == 0
 4        v = 0
 5  else
 6        v = 0
 7        for i = 1 to n
 8              v = max{v, p[i] + CutRod(p, n − i)}
 9  memo[n] = v
10  return v
```

Solution: Compute $\textsc{CutRod}(p, n)$.
Running time: $\Theta(n^2)$.

Constructing Cuts

To construct the cuts we use an array $c[0..n]$ so that $c[n]$ holds the optimal size i of the first piece to cutoff when solving a subproblem of size j.

$\textsc{CutRod}(p[1..n], n)$

```
 1  if memo[n] ≠ ⊥
 2        v = memo[n]
 3  elseif n == 0
 4        v = 0
 5  else
 6        v = 0
 7        for i = 1 to n
 8              if v < p[i] + CutRod(p, n − i)}
 9                    v = p[i] + CutRod(p, n − i)
10                    c = i
11  memo[n] = v
12  c[n] = c
13  return v
```

Coin-Collecting Robot [2]

Input: An $m \times n$ board with no more than one coin at each cell and a coin-collecting robot in the upper left-hand corner of the board. The movement of the robot is subject to the following three rules:

1. The robot must stop in the bottom right corner of the board.
2. The robot can move at most on cell right or one cell down in a single step.
   - Visiting a cell that contains a coin results in the robot collecting the coin.

Output: The maximum number of coins the robot can collect and the path required to collect the coins.

[2] These problems come from "Introduction to the Design and Analysis of Algorithms" by Anany Levitin.

Formulation and Localization

- Formulation: Let $CC(i, j)$ denote the largest number of coins the robot can collect when it visits cell (i, j).
  - There are mn subproblems.
  - Want to compute $CC(m, n)$.
- Localization:
  - The robot can reach the cell (i, j) in one of two possible ways:
    1. From cell $(i-1, j)$ (the cell above)
    2. From cell $(i, j-1)$ (the cell to the left)
  - Let $c_{ij} = 1$ if there is a coin in cell (i, j) and 0 otherwise. Then for each (i, j) with $i \in [0, m]$ and $j \in [0, n]$,

$$CC(i,j) = \begin{cases} \max\{CC(i-1,j), CC(i,j-1)\} + c_{ij}, & \text{If } i > 0 \text{ and } j > 0, \\ 0, & \text{If } i = 0 \text{ or } j = 0. \end{cases}$$

Memoization

Row-wise memoization using $memo[1..m, 1..n]$:

$\mathrm{CC}(i, j, C[1..m, 1..n])$

```
 1  if memo[i,j] ≠ ⊥
 2        v = memo[i,j]
 3  elseif 1 ≤ i ≤ m and 1 ≤ j ≤ n
 4        v = max{CC(i − 1, j, C), CC(i, j − 1, C)} + C[i,j]
 5  elseif i == 0 and 1 ≤ j ≤ n
 6        v = 0
 7  elseif j == 0 and 1 ≤ i ≤ m
 8        v = 0
 9  memo[i,j] = v
10  return v
```

Solution: Compute $\mathrm{CC}(m, n)$.
Running time: $\Theta(mn)$.

Constructing the Path

To find the path we work backward from cell (m, n).

- If $memo[i-1, j] > memo[i, j-1]$ then the path to (i, j) came from above.
- If $memo[i-1, j] < memo[i, j-1]$ then the path to (i, j) came from the left.
- If $memo[i-1, j] = memo[i, j-1]$ then either direction is optimal (and thus valid).

When a tie occurs we simply select any one of them. We can recover the path in $\Theta(n+m)$ time.

Subproblem Graphs

- Subproblems relate to each other and form an acyclic graph.
- A subproblem graph is a graph $G = (V, E)$:
  - V is the set of subproblems, one vertex per subproblem.
  - E is a set of edges such that an edge $(u, v) \in V$ iff subproblem u depends on subproblem v.
- We can think of a dynamic program with memoization as a depth-first search of the subproblem graph.
- Subproblem graphs aid in determining running time for dynamic programs.
  - The size of V is the total number of subproblems.
  - The time to compute a solution to a subproblem is proportional to the out-degree of the subproblem.
  - The total running time of a dynamic program is proportional to the number of vertices and edges.

Bottom Up

- The bottom-up approach is a common technique to remove recursion. (Note: Unlike the memoization approach, there is no recursion in the bottom-up approach.)
- In the *bottom-up* approach we use a table to fill in values in the order of subproblem dependency.
  - i.e., perform a topological sort of the subproblem graph according to the size of the subproblems.
  - Solve the smaller subproblem first.
- Filling in tables dynamically was considered programming before the computer era, hence the name of dynamic programming.

Fibonacci Bottom Up

FIB(n)

```
1  Allocate F[1 .. n]
2  for i = 1 to n
3        if i ≤ 2
4              F[i] = 1
5        else
6              F[i] = F[i − 1] + F[i − 2]
7  return F[n]
```

Rod Cutting Bottom Up

```
CutRod(p[1..n], n)
1  Allocate r[0..n]
2  r[0] = 0
3  for j = 1 to n
4        v = 0
5        for i = 1 to j
6              v = max{v, p[i] + r[j − i]}
7        r[j] = v
8  return r
```

Rod Cutting Bottom Up with Path Construction

EXTENDED-CUTROD($p[1..n], n$)

```
 1  Allocate r[0..n] and c[0..n]
 2  r[0] = 0
 3  for j = 1 to n
 4        v = 0
 5        for i = 1 to j
 6              if v < p[i] + r[j − i]}
 7                    v = p[i] + r[j − i]
 8                    c[j] = i
 9              r[j] = v
10        return r and c
```

Longest Common Subsequence (LCS)

- Let X and Y be two sequences over the same alphabet.
  - A sequence σ is a subsequence of X if there exists a strictly increasing sequence of indices of X: $i_1, i_2, \ldots, i_k$, such that $\sigma_j = x_{i_j}$ for all $j = 1, 2, \ldots, k$.
  - A subsequence σ is common to X and Y if σ is a subsequence of X and σ is a subsequence of Y.
- LCS is defined as follows:
  Input: Two sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$.
  Output: The length of the longest common subsequence.

Formulation and Localization

- Formulation: Let $LCS(i, j)$ denote the length of the longest common subsequence of sequences $X_i = \langle x_1, x_2, \ldots, x_i \rangle$ and $Y_j = \langle y_1, y_2, \ldots, y_j \rangle$.
  - Want to compute $LCS(m, n)$.
  - There are mn subproblems.
- Localization: $LCS(i, j)$ is
  1. Based on $LCS(i-1, j-1)$, i.e., element $x_i = y_j$ and so it should be in the longest common subsequence.
  2. Based on $LCS(i-1, j)$, i.e., $x_i \neq y_j$ and x_i is not in the longest common subsequence.
  3. Based on $LCS(i, j-1)$, i.e., $x_i \neq y_j$ and y_j is not in the longest common subsequence.

$$LCS(i,j) = \begin{cases} 0, & \text{If } i = 0 \text{ or } j = 0, \\ LCS(i-1, j-1) + 1, & \text{If } i, j > 0 \ \& \ x_i = y_j, \\ \max\{LCS(i-1, j), LCS(i, j-1)\}, & \text{If } i, j > 0 \ \& \ x_i \neq y_j. \end{cases}$$

Memoization

$\mathrm{LCS}(i, j, X[1..m], Y[1..n])$

```
1  if memo[i, j] ≠ ⊥
2        v = memo[i, j]
3  elseif i = 0  or j = 0
4        v = 0
5  elseif X[i] == Y[j]
6        v = LCS(i − 1, j − 1, X, Y) + 1
7  else
8        v = max{LCS(i − 1, j, X, Y), LCS(i, j − 1, X, Y)}
9  memo[i, j] = v
10 return v
```

Running time: $\Theta(mn)$.

Bottom Up for LCS Construction

LCS-LENGTH(X, Y)

```
1   INITIALIZATION(X, Y, c, b)
2   for i = 1 to m
3       for j = 1 to n
4           if x_i == y_j
5               c[i, j] = c[i − 1, j − 1] + 1
6               b[i, j] = 1
7           elseif c[i − 1, j] ≥ c[i, j − 1]
8               c[i, j] = c[i − 1, j]
9               b[i, j] = 2
10          else c[i, j] = c[i − 1, j]
11              b[i, j] = 3
12  return c and b
```

INITIALIZATION(X, Y, c, b)

```
1  m = X.length
2  n = Y.length
3  Allocate b[1..m, 1..n] and c[0..m, 0..n]
4  for i = 1 to m
5        c[i, 0] = 0
6  for j = 0 to n
7        c[0, j] = 0
```

LCS Construction

PRINT-LCS(b, X, i, j)

```
1  if i == 0  or j == 0
2        return
3  if b[i, j] == 1
4        PRING-LCS(b, X, i − 1, j − 1)
5        print x_i
6  elseif b[i, j] == 2
7        PRING-LCS(b, X, i − 1, j)
8  else PRING-LCS(b, X, i, j − 1)
```

Resizing Images by Seam Carving

Imagine you want to resize an image by changing its width (with its height fixed) without having it looked distorted.

- Working from the top row of pixels to the bottom: Remove pixels that are neighbors to the pixels below them subject to the constraint that their colors are similar to one of the neighbors in question
- This top row to bottom row sequence of pixels is referred to as a *seam*.

This idea is called seam carving and is from a SIGGRAPH '07 paper.

- Avidan, Shai, and Ariel Shamir. "Seam carving for content-aware image resizing." ACM Transactions on graphics (TOG). Vol. 26. No. 3. ACM, 2007.
- Video can be found at `https://www.youtube.com/watch?v=6NcIJXTlugc`

Definition

Suppose that we have

- An $m \times n$ matrix **A** of RGB values. Each location (i, j) in the array is called a *pixel*. (A pixel is an acronym for picture element.)
- An $m \times n$ matrix **D** of real values called *disruption values* (also referred to as energy).
  - Removing pixels with lower disruption values (i.e., lower energy) is better.

  (Remark: (1) The matrix **D** is computed from **A** using an energy method, such as the dual gradient energy measure (see the next slide). (2) No single measure works well for all images.)

Formalize the seam carving problem as follows:
Input: An $m \times n$ matrix **A** that represents the image and the corresponding matrix **D** representing the disruption measure.
Output: The total disruption measure of the least disruptive seam (and later the seam itself).

The Dual Gradient Energy Measure

- Let (a, b) be a pixel with RGB value $(r_{a,b}, b_{a,b}, b_{a,b})$.
- Let $R_x(a, b)$, $G_x(a, b)$, and $B_x(a, b)$ denote, respectively, the absolute value in differences of red, green, and blue components between pixel $(a + 1, b)$ and pixel $(a - 1, b)$. For example, $R_x(a, b) =\mid r_{a+1,b} - r_{a-1,b} \mid$.
- Let $R_y(a, b)$, $G_y(a, b)$, and $B_y(a, b)$ denote, respectively, the absolute value in differences of red, green, and blue components between pixel $(a, b + 1)$ and pixel $(a, b - 1)$. For example, $G_y(a, b) =\mid g_{a,b+1} - g_{a,b-1} \mid$.
- To calculate the energy of pixels at the border of the image, replace the missing pixel with the pixel from the opposite edge. For example, $B_x(0, 0) =\mid b_{1,0} - b_{m-1,0} \mid$.
- Define $D(a, b) =\mid \Delta_x^2(a, b) \mid + \mid \Delta_y^2(a, b) \mid$, where

$$\begin{aligned} \Delta_x^2(a, b) &= R_x^2(a, b) + G_x^2(a, b) + B_x^2(a, b), \\ \Delta_y^2(a, b) &= R_y^2(a, b) + G_y^2(a, b) + B_y^2(a, b). \end{aligned}$$

Dual Gradient Energy Example

- Let

$$\mathbf{A} = \begin{bmatrix} (255, 100, 55) & (255, 100, 150) & (255, 100, 255) \\ (255, 150, 55) & (255, 150, 153) & (255, 155, 255) \\ (255, 200, 55) & (255, 200, 150) & (255, 200, 255) \end{bmatrix}$$

- There are 8 border pixels and one non-border pixel (1,1).

$$\begin{aligned} R_x(1,1) &= 255 - 255 = 0, \\ G_x(1,1) &= 200 - 100 = 100, \\ B_x(1,1) &= 150 - 150 = 0. \end{aligned}$$

$\Delta_x^2(1,1) = 100^2 = 10000.$

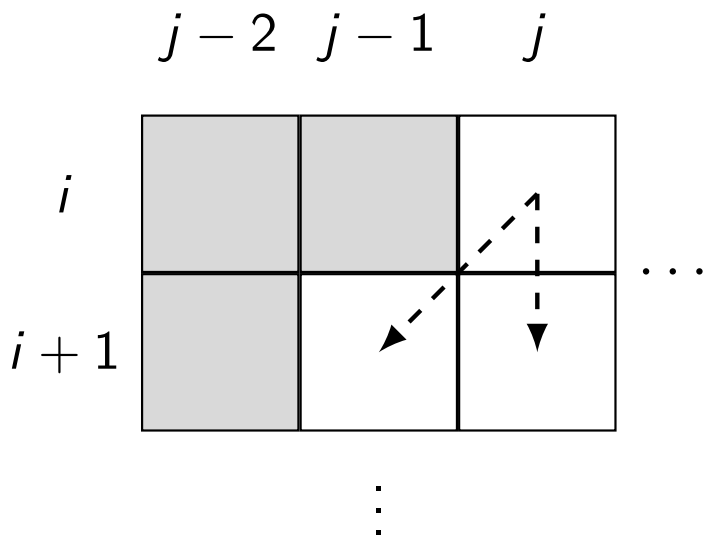
- Likewise, we can compute $\Delta_y^2(1,1) = 5^2 + 200^2 = 40025$.
- $D(1,1) = 10000 + 40025 = 50025$.
- $D(0,0) = 0^2 + 50^2 + 0^2 + 0^2 + 0^2 + 105^2 = 10025 + 2500 = 12525$.

Formulation and Localization

- Formulation: Let $S(i, j)$ denote the disruption of the optimal seam rooted at pixel (i, j). In the worst case an $m \times n$ image requires to solve mn subproblems.
  - We only need to compute a subseam once.
- Localization: The minimum disruption value for any subseam root $S(i, j)$ can be determine based on one of the three lower neighbors in the row below the current row.
  - Idea: use the minimum disruptive seam rooted at $(i - 1, k)$ for $k = j - 1, j$, or $j + 1$ and add the disruption factor of the current pixel.
  - There are four distinct cases we need to consider when trying to determine $S(i, j)$.

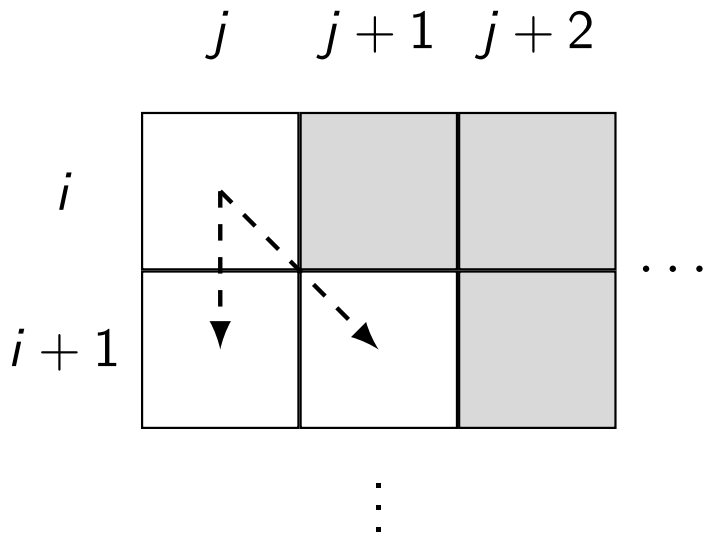
Localization 1

- The current pixel (i, j) is in the bottom row of $\mathbf{A}$.
  - This is the base case; just the disruption value at location (i, j).
- The current pixel (i, j) is in the right-most column and therefore only has the following neighbors: below left or directly below.



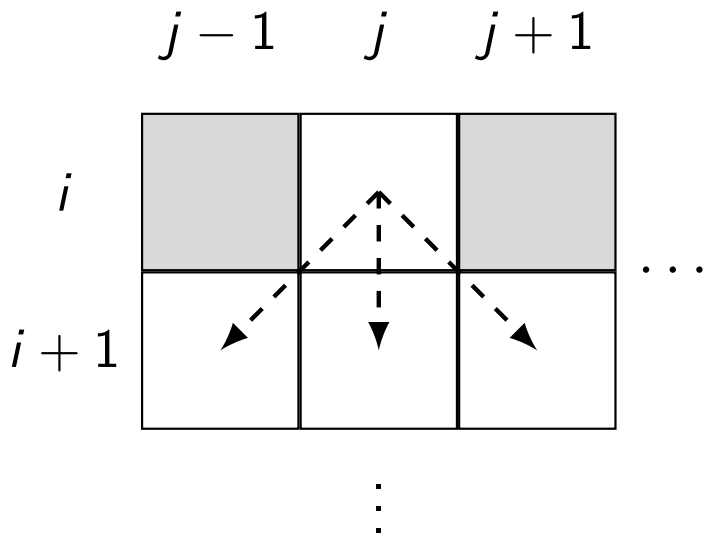
Localization 2

- The current pixel (i, j) is in the left-most column and therefore only has the following neighbors: below right or directly below.



Localization 3

- The current pixel (i, j) is somewhere between the left-most and right-most column of a given row. It has three neighbors: below left, below right, or directly below.



Recurrence

The recurrence relation that handles all these cases is as follows:

$$S(i,j) = \begin{cases} d_{ij} & \text{If } i = m, \\ d_{ij} + \min\{S(i+1,j), S(i+1,j-1)\} & \text{If } j = n \text{ and } i < m, \\ d_{ij} + \min\{S(i+1,j), S(i+1,j+1)\} & \text{If } j = 1 \text{ and } i < m, \\ d_{ij} + \min\{S(i+1,j), S(i+1,j-1), & \text{Otherwise.} \\ \qquad S(i+1,j+1)\} & \end{cases}$$

Solution: Compute $\min_{1 \le j \le n}\{S(1,j)\}$.

- We will assume that there exists a global memo pad $memo[1\,..\,m, 1\,..\,n]$ with every entry initialized to $\perp$.
- We will require two algorithms $\textsc{RSeam}$ that returns the disruption of the best seam rooted at location (i,j) and $\textsc{OSeam}$ which returns the best seam in the image (the solution).

Memoization

$\textsc{RSeam}(i, j, D[1..m, 1..n])$

```
1   if memo[i,j] ≠ ⊥
2         v = memo[i,j]
3   elseif i == m
4         v = D[i,j]
5   elseif i < m and j == n
6         v = D[i,j] + min{RSeam(i+1,j,D), RSeam(i+1,j-1)}
7   elseif i < m and j == 1
8         v = D(i,j) + min{RSeam(i+1,j,D), RSeam(i+1,j+1)}
9   else
10        v = D(i,j) + min {RSeam(i+1,j-1,D), RSeam(i+1,j,D),
                RSeam(i+1,j+1)}
11  memo[i,j] = v
12  return v
```

Running time: $\Theta((n-i)(m-j))$ for the operations.

Optimal Seam Construction

$\text{OSeam}(D[1\,..\,m, 1\,..\,n])$

```
1  min = ∞
2  for j = 1 to n
3        v = RSeam(1, j, D)
4        if v < min
5              min = v
6  return min
```