Dynamic Programming

Dor Shmoish

What is DP?

- DP is an **algorithm design method** which solves problems by **combining solutions to subproblems**.
- Dynamic programming applies when **subproblems overlap**, i.e. the subproblems share subsubproblems.
- A DP algorithm solves each subsubproblem **just once** and then **saves its answer in a table**.
- DP is typically applied to **optimization problems**.

Recognizing DP

DP is applicable to a given problem when:

1. The solution consists of making a **choice**, which leads to one or more subproblems.
2. The space of subproblems is not too big.
   - **Subsubproblems overlap.**
3. Solutions to subproblems within an optimal solution to the original problem are themselves optimal.

Developing an Algorithm

We follow a sequence of four steps:

1. Characterize the structure of an optimal solution.
2. Define **recursively** the value of an optimal solution.
3. Compute the value of an optimal solution.
   - Typically in a bottom-up fashion.
4. Construct an optimal solution from the computed information.

Number Sequence Game

- There are N cards on the table, arranged in a row. Positive integers $a_1, \ldots, a_N$ are written on the cards.
- Two players play a game taking turns.
- At each turn, the current player takes <u>one</u> card: either the **leftmost** or **rightmost** card.

Number Sequence Game

- The game ends after exactly N turns, when all cards have been taken.
- The goal of each player is to **maximize their sum**.
- Assuming both players play optimally, what are **their sums at the end of the game**?

Number Sequence Game

- <u>Example</u>

1	2	10	3

Player 1:

Player 2:

Solution

- <u>Subproblems:</u>

  A game $\boldsymbol{G_{i,j}}$ with cards $\boldsymbol{a_i}, \dots, \boldsymbol{a_j}$ $(1 \le i \le j \le N)$.

- Let

  $$\boldsymbol{f(i,j) = \textbf{sum of player 1 in game } G_{i,j}}$$

  for all $1 \le i \le j \le N$.

- <u>Note:</u> $\boldsymbol{\textbf{sum of player 2} = a_i + \cdots + a_j - f(i,j)}$.

Solution

- Recursive formula:

$$f(i,j) = a_i + \cdots + a_j - \min \begin{cases} f(i+1,j) \\ f(i,j-1) \end{cases}$$

- Base cases: $f(i,i) = a_i$.
- "Plain" recursion: exponential running time.
  - Each problem is split into 2 subproblems.
- Dynamic programming: $O(N^2)$ running time!

Implementation

- Maintain a matrix $f[1 \dots n][1 \dots n]$.
  - Each entry $f[i][j]$ will hold the value $f(i, j)$.
- Initialize the entries $f[i][i]$ with a_i.
- Update the entries $f[i][j]$ for increasing $\boldsymbol{j - i}$ using the recursive formula and the previous entries.
  - For $f[i][j]$ we need $f[i + 1][j]$ and $f[i][j - 1]$, both of which have a smaller $\boldsymbol{j - i}$.

Edit Distance

- Let $A = a_1 a_2 \dots a_n$ and $B = b_1 b_2 \dots b_m$ be two strings. We'd like to change A to B character by character.
- We allow three types of changes (or edit steps):
  1) insert – insert a character into the string.
  2) delete – delete a character from the string.
  3) replace – replace one character with a different one.
- What is the **edit distance** $d_{\text{edit}}(A, B)$, i.e. minimum number of edit steps required to get from A to B?

Edit Distance

- Example

$$A = abbc,\ B = babb$$

- Possibility 1:

$$abbc \xrightarrow{\text{delete}} bbc \xrightarrow{\text{insert}} babc \xrightarrow{\text{replace}} babb$$

- Possibility 2:

$$abbc \xrightarrow{\text{delete}} abb \xrightarrow{\text{insert}} babb$$

Edit Distance

- Use cases:
  - Revisions maintenance (Git).
  - File comparisons (archiving several similar versions).

Solution

- Subproblems:

$$\boldsymbol{A^{(i)} = a_1 a_2 \ldots a_i,\ B^{(i)} = b_1 b_2 \ldots b_i}$$

- Let

$$\boldsymbol{d(i,j) = d_{edit}\left(A^{(i)}, B^{(j)}\right)}$$

for all $1 \leq i \leq n, 1 \leq j \leq m$.

- **Goal:** Find the value of $d(n,m)$ using the values of $d(i,j)$ for smaller i,j.

Solution

- We want to match the last characters a_n and b_m.

  We can either:
  - insert b_m: $\boldsymbol{d(n, m-1)+1}$ steps.
  - delete a_n: $\boldsymbol{d(n-1, m)+1}$ steps.
  - replace a_n by b_m: $\boldsymbol{d(n-1, m-1)+1}$ steps ($a_n \neq b_m$).
  - match a_n with b_m: $\boldsymbol{d(n-1, m-1)}$ steps ($a_n = b_m$).

- Recursive formula:

$$\boldsymbol{d(n,m) = \min \begin{cases} d(n, m-1) + 1 \\ d(n-1, m) + 1 \\ d(n-1, m-1) + \delta(a_n, b_m) \end{cases}}$$

Solution

- Recursive formula:

$$\boldsymbol{d(n,m)} = \mathbf{min}\begin{cases} \boldsymbol{d(n,m-1)+1} \\ \boldsymbol{d(n-1,m)+1} \\ \boldsymbol{d(n-1,m-1)+\delta(a_n,b_m)} \end{cases}$$

- Base cases: $\boldsymbol{d(i,0)=i}$ and $\boldsymbol{d(0,j)=j}$.
- "Plain" recursion: exponential running time.
  - Each problem is split into 3 subproblems.
- Dynamic programming: $\boldsymbol{O(nm)}$ running time!

Implementation

- Maintain a matrix $d[1 \ldots n][1 \ldots m]$.
  - Each entry $d[i][j]$ will hold the value $d(i, j)$.
- Initialize the entries $d[i][0]$ and $d[0][j]$ according to the initial conditions.
- Update the entries $d[i][j]$ for increasing i and j using the recursive formula and the previous entries.
- Exercise: What if we wanted to find an optimal **sequence** of edits, and not just the edit distance?

Pseudocode

```
for i := 0 to n do
    d[i][0] := i;
for j := 1 to m do
    d[0][j] := j;
for i := 1 to n do
    for j := 1 to m do
        x := d[i-1][j] + 1;
        y := d[i][j-1] + 1;
        if A[i] = B[j] then
            z := d[i-1][j-1];
        else
            z := d[i-1][j-1] + 1;
        d[i][j] := min(x, y, z);
```

Questions?

- *Due to time limitations, only $O(1)$ questions will be answered.*