Agenda

We've done

- Greedy Method
- Divide and Conquer

Now

- Designing Algorithms with the Dynamic Programming Method

Outline

A Quote from Richard Bellman

"Eye of the Hurricane: An Autobiography"

I spent the Fall quarter (of 1950) at RAND. My first task was to find a name for multistage decision processes. An interesting question is, Where did the name, dynamic programming, come from? The 1950s were not good years for mathematical research. We had a very interesting gentlemen in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word, research. ... I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. ... Thus, I thought dynamic programming was a good name. It was something not even a Congressmann could object to. So I used it as an umbrella for my activities.

A General Description

A General Description

1. Identify the sub-problems
   - Often sub-problems share subsub-problems
   - Total number of $(\text{sub})^i$-problems is "small" (a polynomial number)

A General Description

1. Identify the sub-problems
   - Often sub-problems share subsub-problems
   - Total number of $(\text{sub})^i$-problems is "small" (a polynomial number)
2. Write a recurrence for the objective function: solution to a problem can be computed from solutions to sub-problems
   - Be careful with the base cases

A General Description

1. Identify the sub-problems
   - Often sub-problems share subsub-problems
   - Total number of $(\text{sub})^i$-problems is "small" (a polynomial number)
2. Write a recurrence for the objective function: solution to a problem can be computed from solutions to sub-problems
   - Be careful with the base cases
3. Investigate the recurrence to see how to use a "table" to solve it

A General Description

1. Identify the sub-problems
   - Often sub-problems share subsub-problems
   - Total number of $(\text{sub})^i$-problems is "small" (a polynomial number)
2. Write a recurrence for the objective function: solution to a problem can be computed from solutions to sub-problems
   - Be careful with the base cases
3. Investigate the recurrence to see how to use a "table" to solve it
4. Design appropriate data structure(s) to construct an optimal solution

A General Description

1. Identify the sub-problems
    - Often sub-problems share subsub-problems
    - Total number of $(\text{sub})^i$-problems is "small" (a polynomial number)
2. Write a recurrence for the objective function: solution to a problem can be computed from solutions to sub-problems
    - Be careful with the base cases
3. Investigate the recurrence to see how to use a "table" to solve it
4. Design appropriate data structure(s) to construct an optimal solution
5. Pseudo code

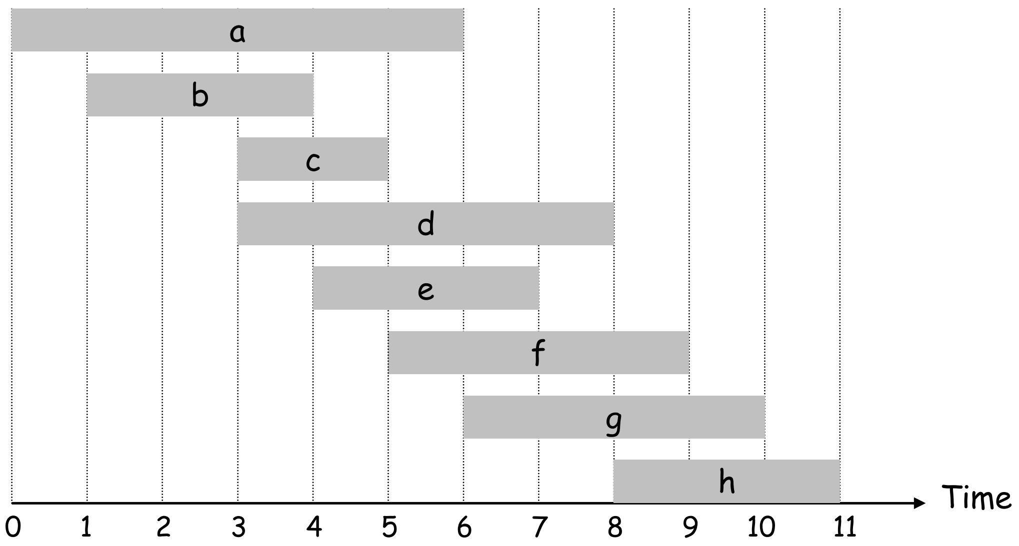
A General Description

1. Identify the sub-problems
    - Often sub-problems share subsub-problems
    - Total number of $(\text{sub})^i$-problems is "small" (a polynomial number)
2. Write a recurrence for the objective function: solution to a problem can be computed from solutions to sub-problems
    - Be careful with the base cases
3. Investigate the recurrence to see how to use a "table" to solve it
4. Design appropriate data structure(s) to construct an optimal solution
5. Pseudo code
6. Analysis of time and space

Outline

Weighted Interval Scheduling: Problem Definition

- Each interval I_j now has a weight $w_j \in \mathbb{Z}^+$
- Find non-overlapping intervals with maximum total weight

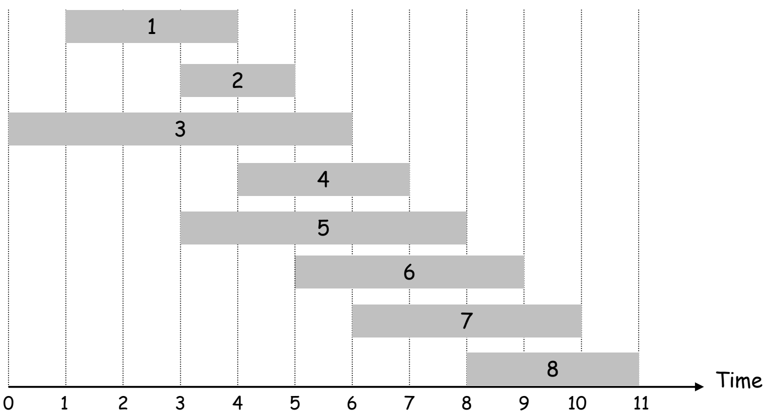


The Structure of an Optimal Solution

The Structure of an Optimal Solution

- Order intervals so that $f_1 \leq f_2 \leq \cdots \leq f_n$
- For each j, let $p(j)$ be the largest index $i < j$ such that I_i and I_j do not overlap; $p(j) = 0$ if no such i

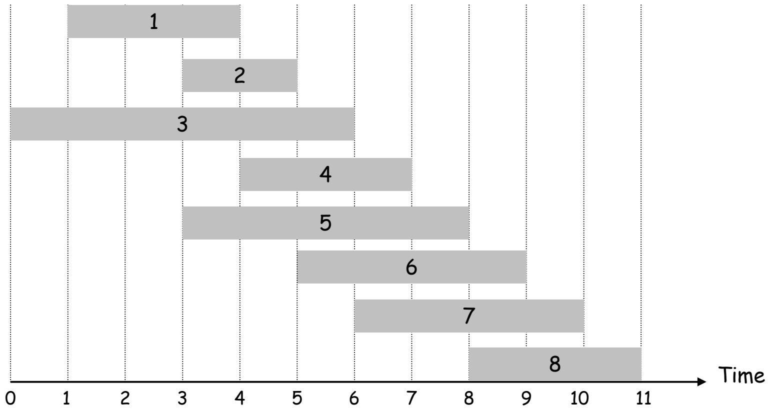
p(1)=0
p(2)=0
p(3)=0
p(4)=1
p(5)=0
p(6)=2
p(7)=3
p(8)=5



The Structure of an Optimal Solution

- Order intervals so that $f_1 \le f_2 \le \cdots \le f_n$
- For each j, let $p(j)$ be the largest index $i < j$ such that I_i and I_j do not overlap; $p(j) = 0$ if no such i

p(1)=0
p(2)=0
p(3)=0
p(4)=1
p(5)=0
p(6)=2
p(7)=3
p(8)=5

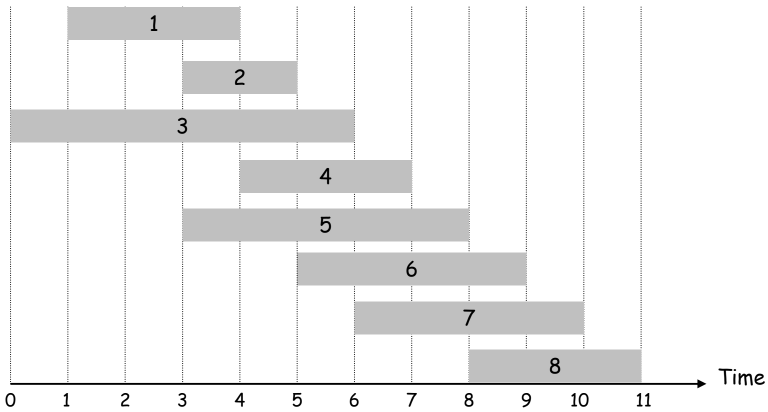


- Let $\mathcal{O}$ be any optimal solution

The Structure of an Optimal Solution

- Order intervals so that $f_1 \le f_2 \le \cdots \le f_n$
- For each j, let $p(j)$ be the largest index $i < j$ such that I_i and I_j do not overlap; $p(j) = 0$ if no such i

p(1)=0
p(2)=0
p(3)=0
p(4)=1
p(5)=0
p(6)=2
p(7)=3
p(8)=5



- Let $\mathcal{O}$ be any optimal solution
  - If $I_n \in \mathcal{O}$, then $\mathcal{O}' = \mathcal{O} - \{I_n\}$ must be optimal for $\{I_1, \dots, I_{p(n)}\}$
  - Else $I_n \notin \mathcal{O}$, then $\mathcal{O}$ must be optimal for $\{I_1, \dots, I_{n-1}\}$

The Recurrence

- Identify subproblems: optimal solution for $\{I_1, \dots, I_n\}$ seems to depend on some optimal solutions to $\{I_1, \dots, I_j\}$, $j = 0..n$
- For $j \le n$, let $\textsc{opt}(j)$ be the cost of an optimal solution to $\{I_1, \dots, I_j\}$
- Crucial Observation:

$$\textsc{opt}(j) = \begin{cases} \max\{w_j + \textsc{opt}(p(j)), \textsc{opt}(j-1)\} & j \ge 1 \\ 0 & j = 0 \end{cases}$$

The Recurrence

- Identify subproblems: optimal solution for $\{I_1, \dots, I_n\}$ seems to depend on some optimal solutions to $\{I_1, \dots, I_j\}$, $j = 0..n$
- For $j \le n$, let $\textsc{opt}(j)$ be the cost of an optimal solution to $\{I_1, \dots, I_j\}$
- Crucial Observation:

$$\textsc{opt}(j) = \begin{cases} \max\{w_j + \textsc{opt}(p(j)), \textsc{opt}(j-1)\} & j \ge 1 \\ 0 & j = 0 \end{cases}$$

Related question

How do we compute the array $p(j)$ efficiently?

First Attempt at Implementing the Idea

$\textsc{Compute-Opt}(j)$

```
if j ≤ 0 then
    Return 0
else
    Return max{w_j + Compute-Opt(p(j)), Compute-Opt(j − 1)}
end if
```


First Attempt at Implementing the Idea

$\textsc{Compute-Opt}(j)$

```
if j ≤ 0 then
    Return 0
else
    Return max{w_j + Compute-Opt(p(j)), Compute-Opt(j − 1)}
end if
```

Proof of correctness: often not needed, because it can easily be done by induction.

First Attempt at Implementing the Idea

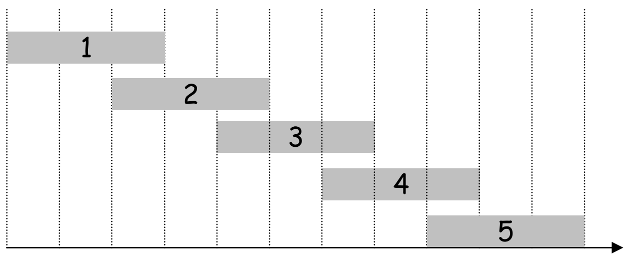
Compute-Opt(j)

if $j \leq 0$ **then**
    Return 0
else
    Return $\max\{w_j + \text{Compute-Opt}(p(j)), \text{Compute-Opt}(j-1)\}$
end if

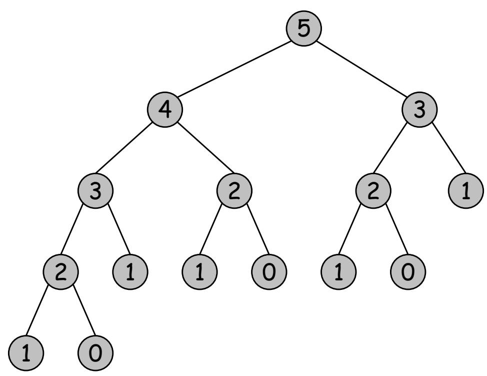
Proof of correctness: often not needed, because it can easily be done by induction. (You do have to justify your recurrence though!)

First Attempt was Bad

- For the same reason FibA was bad.



p(1) = 0, p(j) = j-2



Fixing the Algorithm: a Top-Down Approach

Fixing the Algorithm: a Top-Down Approach

- Key Idea of Dynamic Programming: use a table, in this case an array, to store already computed things

Fixing the Algorithm: a Top-Down Approach

- Key Idea of Dynamic Programming: use a table, in this case an array, to store already computed things
- Use $M[0..n]$ to store $\text{OPT}(0), \dots, \text{OPT}(n)$, initially fill M with -1's

Fixing the Algorithm: a Top-Down Approach

- Key Idea of Dynamic Programming: use a table, in this case an array, to store already computed things
- Use $M[0..n]$ to store $\text{OPT}(0), \dots, \text{OPT}(n)$, initially fill M with -1's

M-COMP-OPT(j)

if $j = 0$ **then**
  Return 0
else if $M[j] \neq -1$ **then**
  Return $M[j]$
else
  $M[j] \leftarrow \max\{w_j + \text{M-COMP-OPT}(p(j)), \text{M-COMP-OPT}(j-1)\}$
  Return $M[j]$
end if

- The top-down approach is often called memoization
- Running time: $O(n)$.

©Hung Q. Ngo (SUNY at Buffalo) CSE 531 Algorithm Analysis and Design 11 / 74

Fixing the Algorithm: a Bottom-Up Approach

Comp-Opt(j)

```
M[0] ← 0
for j = 1 to n do
    M[j] ← max{w_j + M[p(j)], M[j − 1]}
end for
```


Fixing the Algorithm: a Bottom-Up Approach

Comp-Opt(j)

```
M[0] ← 0
for j = 1 to n do
    M[j] ← max{w_j + M[p(j)], M[j − 1]}
end for
```


Bottom-Up vs Top-Down

- Bottom-Up solves all subproblems, Top-Down only solves necessary sub-problems
- Bottom-Up does not involve many function calls, and thus often is faster

Constructing an Optimal Schedule

CONSTRUCT-SOLUTION(j)

1: **if** $j = 0$ **then**
2: Return $\emptyset$
3: **else if** $w_j + M[p(j)] \geq M[j-1]$ **then**
4: Return CONSTRUCT-SOLUTION$(p(j)) \cup \{I_j\}$
5: **else**
6: Return CONSTRUCT-SOLUTION$(p(j-1))$
7: **end if**

Outline

Longest Common Subsequence: Problem Definition

X	=	t	h	i	s	i	s	c	r	a	z	y
Z	=		h	i				c		a	z	y

Z is a subsequence of X.

X	=	**t**	h	**i**	**s**	**i**	s	c	r	a	z	y		
Y	=	b	u	**t**	**i**	n	t	e	r	e	**s**	t	**i**	n	g

So, $Z = [t, i, s, i]$ is a common subsequence of X and Y

The Problem

Given 2 sequences X and Y of lengths m and n, respectively, find a common subsequence Z of longest length

The Structure of an Optimal Solution

- Denote $X = [x_1, \dots, x_m]$, $Y = [y_1, \dots, y_n]$
- Key observation: let $\textsc{lcs}(X, Y)$ be the length of an LCS of X and Y

The Structure of an Optimal Solution

- Denote $X = [x_1, \dots, x_m]$, $Y = [y_1, \dots, y_n]$
- Key observation: let $\textsc{lcs}(X, Y)$ be the length of an LCS of X and Y
  - If $x_m = y_n$, then

$$\textsc{lcs}(X, Y) = 1 + \textsc{lcs}([x_1, \dots, x_{m-1}], [y_1, \dots, y_{n-1}])$$

The Structure of an Optimal Solution

- Denote $X = [x_1, \dots, x_m]$, $Y = [y_1, \dots, y_n]$
- Key observation: let $\textsc{lcs}(X, Y)$ be the length of an LCS of X and Y
  - If $x_m = y_n$, then

$$\textsc{lcs}(X, Y) = 1 + \textsc{lcs}([x_1, \dots, x_{m-1}], [y_1, \dots, y_{n-1}])$$

  - If $x_m \neq y_n$, then either

$$\textsc{lcs}(X, Y) = \textsc{lcs}([x_1, \dots, x_m], [y_1, \dots, y_{n-1}])$$

or

$$\textsc{lcs}(X, Y) = \textsc{lcs}([x_1, \dots, x_{m-1}], [y_1, \dots, y_n])$$

The Recurrence

The Recurrence

- For $0 \le i \le m$, $0 \le j \le n$, let

$$\begin{aligned} X_i &= [x_1, \dots, x_i] \\ Y_j &= [y_1, \dots, y_j] \end{aligned}$$

The Recurrence

- For $0 \leq i \leq m$, $0 \leq j \leq n$, let

$$\begin{aligned} X_i &= [x_1, \dots, x_i] \\ Y_j &= [y_1, \dots, y_j] \end{aligned}$$

- Let $c[i, j] = \text{LCS}[X_i, Y_j]$, then

$$c[i,j] = \begin{cases} 0 & \text{if } i \text{ or } j \text{ is } 0 \\ 1 + c[i-1, j-1] & \text{if } x_i = y_j \\ \max(c[i-1, j], c[i, j-1]) & \text{if } x_i \neq y_j \end{cases}$$

The Recurrence

- For $0 \leq i \leq m$, $0 \leq j \leq n$, let

$$\begin{aligned} X_i &= [x_1, \dots, x_i] \\ Y_j &= [y_1, \dots, y_j] \end{aligned}$$

- Let $c[i, j] = \text{LCS}[X_i, Y_j]$, then

$$c[i,j] = \begin{cases} 0 & \text{if } i \text{ or } j \text{ is } 0 \\ 1 + c[i-1, j-1] & \text{if } x_i = y_j \\ \max(c[i-1, j], c[i, j-1]) & \text{if } x_i \neq y_j \end{cases}$$

- Hence, $c[i, j]$ in general depends on one of three entries: the North entry $c[i-1, j]$, the West entry $c[i, j-1]$, and the NorthWest entry $c[i-1, j-1]$.

Computing the Optimal Value

$\textsc{LCS-Length}(X, Y, m, n)$

$c[i, 0] \leftarrow 0, \ \forall i = 0, \dots, m; \quad c[0, j] \leftarrow 0, \ \forall j = 0, \dots, n;$
for $i \leftarrow 1$ **to** m **do**
    for $j \leftarrow 1$ **to** n **do**
        if $x_i = y_j$ **then**
            $c[i, j] \leftarrow 1 + c[i-1, j-1];$
        else if $c[i-1, j] > c[i, j-1]$ **then**
            $c[i, j] \leftarrow c[i-1, j];$
        else
            $c[i, j] \leftarrow c[i, j-1];$
        end if
    end for
end for

Construting an Optimal Solution

- Z is a global array, initially empty

$\textsc{LCS-Construction}(Z, i, j)$

$k \leftarrow c[i, j]$
if $i = 0$ or $j = 0$ **then**
    Return Z
else if $x_i = y_j$ **then**
    $Z[k] \leftarrow x_i$
    $\textsc{LCS-Construction}(i-1, j-1)$
else if $c[i-1, j] > c[i, j-1]$ **then**
    $\textsc{LCS-Construction}(i-1, j)$
else
    $\textsc{LCS-Construction}(i, j-1)$
end if

Time and Space Analysis

- Filling out the c table takes $\Theta(mn)$-time, which is also the running time of LCS-Length
- The space requirement is also $\Theta(mn)$
- LCS-Construction takes $O(m+n)$ (why?)

Outline

Segmented Least Square: Problem Definition

Segmented Least Square: Problem Definition

- Least Squares is a foundational problem in statistics and numerical analysis
- Given n points in the plane: $P = \{(x_1, y_1), \dots, (x_n, y_n)\}$
- Find a line L: $y = ax + b$ that "fits" them best

Segmented Least Square: Problem Definition

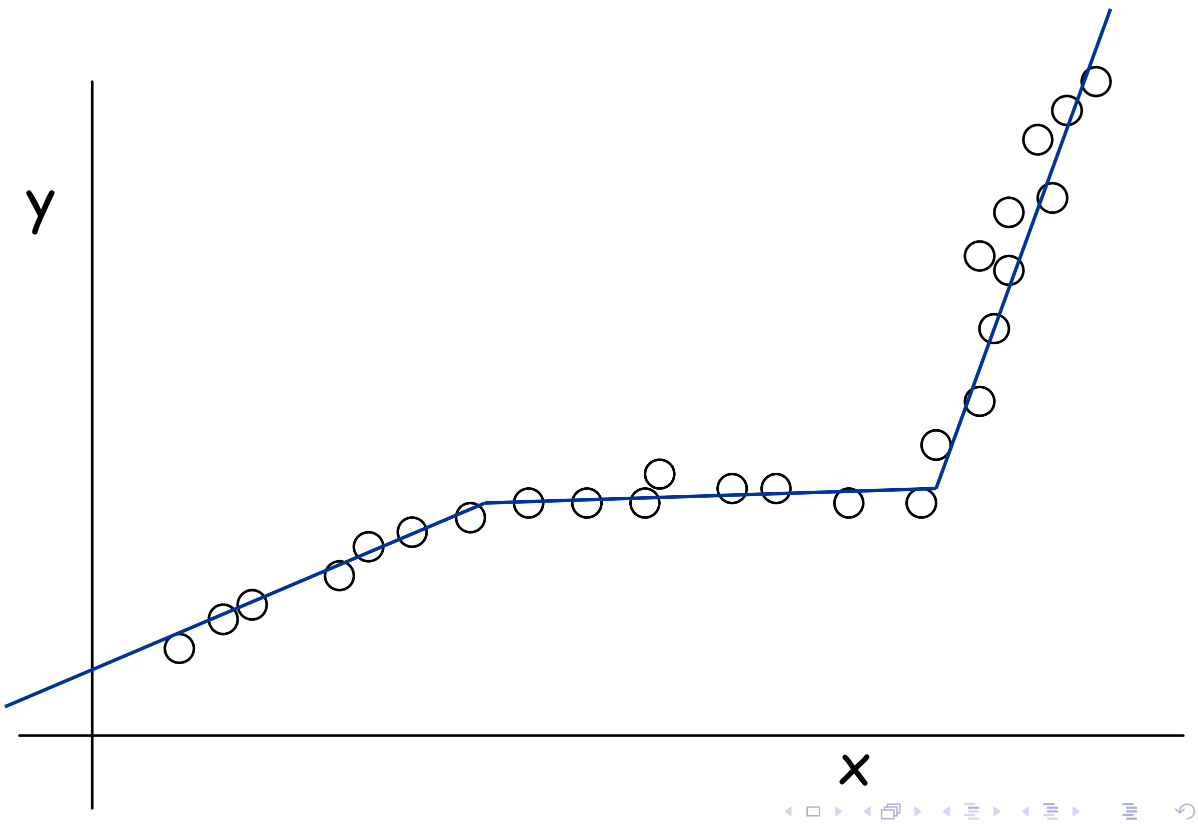
- Least Squares is a foundational problem in statistics and numerical analysis
- Given n points in the plane: $P = \{(x_1, y_1), \dots, (x_n, y_n)\}$
- Find a line L: $y = ax + b$ that "fits" them best
- "Fittest" means minimizing the error term

$$\text{ERROR}(L, P) = \sum_{i=1}^{n} (y_i - ax_i - b)^2$$

- Basic calculus gives

$$a = \frac{n \sum_i x_i y_i - (\sum_i x_i)(\sum_i y_i)}{n \sum_i x_i^2 - (\sum_i x_i)^2} \text{ and } b = \frac{\sum_i y_i - a \sum_i x_i}{n}$$

Practical Issues



A Compromised Objective Function

- Given n points $p_1 = (x_1, y_1), \dots, p_n = (x_n, y_n)$
- $x_1 < x_2 < \cdots < x_n$
- Want to minimize both the number s of segments and total (squared) error e
- A common method: use a weighted sum $e + cs$ for a given constant $c > 0$

A Compromised Objective Function

- Given n points $p_1 = (x_1, y_1), \dots, p_n = (x_n, y_n)$
- $x_1 < x_2 < \cdots < x_n$
- Want to minimize both the number s of segments and total (squared) error e
- A common method: use a weighted sum $e + cs$ for a given constant $c > 0$

More precisely

- Find a partition of the points into some k contiguous parts
- Fit jth part with the best segment with error e_j
- Want to minimize $\sum_{j=1}^{k} e_j + ck$

The Structure of an Optimal Solution

The Structure of an Optimal Solution

- The last part of an optimal solution $\mathcal{O}$ consists of points $p_i, \dots, p_n$ for some $i = 1, \dots, n$

The Structure of an Optimal Solution

- The last part of an optimal solution $\mathcal{O}$ consists of points $p_i, \dots, p_n$ for some $i = 1, \dots, n$
- The cost for segments fitting $p_1, \dots, p_{i-1}$ must be optimal too! Let $\mathcal{O}'$ be an optimal solution to $p_1, \dots, p_{i-1}$

The Structure of an Optimal Solution

- The last part of an optimal solution $\mathcal{O}$ consists of points $p_i, \dots, p_n$ for some $i = 1, \dots, n$
- The cost for segments fitting $p_1, \dots, p_{i-1}$ must be optimal too! Let $\mathcal{O}'$ be an optimal solution to $p_1, \dots, p_{i-1}$
- In English, if $p_i, \dots, p_n$ forms the last part of $\mathcal{O}$, then

$$\text{cost}(\mathcal{O}) = \text{cost}(\mathcal{O}') + e(i, n) + c$$

($e(i, n)$ is the least error of fitting a line through $p_i, \dots, p_n$)

The Recurrence

The Recurrence

- Let $e(i, j)$ be the least error fitting a line through $p_i, p_{i+1}, \dots, p_j$

The Recurrence

- Let $e(i,j)$ be the least error fitting a line through $p_i, p_{i+1}, \dots, p_j$
- Let $\text{OPT}(i)$ be the optimal cost for input $\{p_1, \dots, p_i\}$

The Recurrence

- Let $e(i,j)$ be the least error fitting a line through $p_i, p_{i+1}, \dots, p_j$
- Let $\text{OPT}(i)$ be the optimal cost for input $\{p_1, \dots, p_i\}$
- Then,

$$\text{OPT}(j) = \begin{cases} 0 & \text{if } j = 0 \\ \min\limits_{1 \le i \le j} \{\text{OPT}(i-1) + e(i,j) + c\} & \text{if } j > 0 \end{cases}$$

Pseudo-Code

- Pre-compute $e(i, j)$ for all $i < j$: brute-force takes $O(n^3)$, finer implementation takes $O(n^2)$
- Use recurrence to fill up array $\text{OPT}[0, \dots, n]$, another $O(n^2)$

FIND-SEGMENTS(j)

```
1: if j = 0 then
2:     Return ∅
3: else
4:     Find i minimizing OPT(i − 1) + e(i, j) + c
5:     Return segment {p_i, ..., p_j} and result of FIND-SEGMENTS(i − 1)
6: end if
```


Outline

Matrix Chain Multiplication: Problem Definitions

Given $\mathbf{A}_{10\times100}$, $\mathbf{B}_{100\times25}$, then calculating $\mathbf{AB}$ requires $10\cdot100\cdot25 = 25,000$ multiplications.
Given $\mathbf{A}_{10\times100}$, $\mathbf{B}_{100\times25}$, $C_{25\times4}$, then by associativity

$$\mathbf{ABC} = (\mathbf{AB})\mathbf{C} = \mathbf{A}(\mathbf{BC})$$

Matrix Chain Multiplication: Problem Definitions

Given $\mathbf{A}_{10\times100}$, $\mathbf{B}_{100\times25}$, then calculating $\mathbf{AB}$ requires $10\cdot100\cdot25 = 25,000$ multiplications.
Given $\mathbf{A}_{10\times100}$, $\mathbf{B}_{100\times25}$, $C_{25\times4}$, then by associativity

$$\mathbf{ABC} = (\mathbf{AB})\mathbf{C} = \mathbf{A}(\mathbf{BC})$$

- $\mathbf{AB}$ requires $25,000$ multiplications
- $(\mathbf{AB})\mathbf{C}$ requires $10\cdot25\cdot4 = 1000$ more multiplications
- totally $26,000$ multiplications

Matrix Chain Multiplication: Problem Definitions

Given $\mathbf{A}_{10\times 100}$, $\mathbf{B}_{100\times 25}$, then calculating $\mathbf{AB}$ requires $10\cdot 100\cdot 25 = 25,000$ multiplications.
Given $\mathbf{A}_{10\times 100}$, $\mathbf{B}_{100\times 25}$, $C_{25\times 4}$, then by associativity

$$\mathbf{ABC} = (\mathbf{AB})\mathbf{C} = \mathbf{A}(\mathbf{BC})$$

- $\mathbf{AB}$ requires $25,000$ multiplications
- $(\mathbf{AB})\mathbf{C}$ requires $10\cdot 25\cdot 4 = 1000$ more multiplications
- totally $26,000$ multiplications

On the other hand

- $\mathbf{BC}$ requires $100\cdot 25\cdot 4 = 10,000$ multiplications
- $\mathbf{A}(\mathbf{BC})$ requires $10\times 100\times 4 = 4000$ more multiplications
- totally $14,000$ multiplications

Problem Definitions (cont)

There are 5 ways to parenthesize $\mathbf{ABCD}$:

$$(\mathbf{A}(\mathbf{B}(\mathbf{CD}))), (\mathbf{A}((\mathbf{BC})\mathbf{D})), ((\mathbf{AB})(\mathbf{CD})), ((\mathbf{A}(\mathbf{BC}))\mathbf{D}), (((\mathbf{AB})\mathbf{C})\mathbf{D})$$

In general, given n matrices:

$$\begin{array}{lll} \mathbf{A}_1 & \text{of dimension} & p_0\times p_1 \\ \mathbf{A}_2 & \text{of dimension} & p_1\times p_2 \\ \vdots & \vdots & \vdots \\ \mathbf{A}_n & \text{of dimension} & p_{n-1}\times p_n \end{array}$$

Number of ways to parenthesis $\mathbf{A}_1\mathbf{A}_2\dots\mathbf{A}_n$ is

Problem Definitions (cont)

There are 5 ways to parenthesize $\mathbf{ABCD}$:

$$(\mathbf{A}(\mathbf{B}(\mathbf{CD}))), (\mathbf{A}((\mathbf{BC})\mathbf{D})), ((\mathbf{AB})(\mathbf{CD})), ((\mathbf{A}(\mathbf{BC}))\mathbf{D}), (((\mathbf{AB})\mathbf{C})\mathbf{D})$$

In general, given n matrices:

$$\begin{array}{lll} \mathbf{A}_1 & \text{of dimension} & p_0 \times p_1 \\ \mathbf{A}_2 & \text{of dimension} & p_1 \times p_2 \\ \vdots & \vdots & \vdots \\ \mathbf{A}_n & \text{of dimension} & p_{n-1} \times p_n \end{array}$$

Number of ways to parenthesis $\mathbf{A}_1\mathbf{A}_2 \dots \mathbf{A}_n$ is

$$\frac{1}{n+1}\binom{2n}{n} = \frac{1}{n+1}\frac{(2n)!}{n!n!} = \Omega\left(\frac{4^n}{n^{3/2}}\right)$$

Problem Definitions (cont)

There are 5 ways to parenthesize $\mathbf{ABCD}$:

$$(\mathbf{A}(\mathbf{B}(\mathbf{CD}))), (\mathbf{A}((\mathbf{BC})\mathbf{D})), ((\mathbf{AB})(\mathbf{CD})), ((\mathbf{A}(\mathbf{BC}))\mathbf{D}), (((\mathbf{AB})\mathbf{C})\mathbf{D})$$

In general, given n matrices:

$$\begin{array}{lll} \mathbf{A}_1 & \text{of dimension} & p_0 \times p_1 \\ \mathbf{A}_2 & \text{of dimension} & p_1 \times p_2 \\ \vdots & \vdots & \vdots \\ \mathbf{A}_n & \text{of dimension} & p_{n-1} \times p_n \end{array}$$

Number of ways to parenthesis $\mathbf{A}_1\mathbf{A}_2 \dots \mathbf{A}_n$ is

$$\frac{1}{n+1}\binom{2n}{n} = \frac{1}{n+1}\frac{(2n)!}{n!n!} = \Omega\left(\frac{4^n}{n^{3/2}}\right)$$

The Problem

Find a parenthesization with the least number of multiplications

Structure of an Optimal Solution

Structure of an Optimal Solution

- Suppose we split between $\mathbf{A}_k$ and $\mathbf{A}_{k+1}$, then the parenthesization of $\mathbf{A}_1 \dots \mathbf{A}_k$ and $\mathbf{A}_{k+1} \dots \mathbf{A}_n$ have to also be optimal

Structure of an Optimal Solution

- Suppose we split between $\mathbf{A}_k$ and $\mathbf{A}_{k+1}$, then the parenthesization of $\mathbf{A}_1 \dots \mathbf{A}_k$ and $\mathbf{A}_{k+1} \dots \mathbf{A}_n$ have to also be optimal
- Let $c[1,k]$ and $c[k+1,n]$ be the optimal costs for the subproblems, then the cost of splitting at $k, k+1$ is

$$c[1,k] + c[k+1,n] + p_0 p_k p_n$$

Structure of an Optimal Solution

- Suppose we split between $\mathbf{A}_k$ and $\mathbf{A}_{k+1}$, then the parenthesization of $\mathbf{A}_1 \dots \mathbf{A}_k$ and $\mathbf{A}_{k+1} \dots \mathbf{A}_n$ have to also be optimal
- Let $c[1,k]$ and $c[k+1,n]$ be the optimal costs for the subproblems, then the cost of splitting at $k, k+1$ is

$$c[1,k] + c[k+1,n] + p_0 p_k p_n$$

- Thus, the main recurrence is

$$c[1,n] = \min_{1 \le k < n} \left(c[1,k] + c[k+1,n] + p_0 p_k p_n \right)$$

Structure of an Optimal Solution

- Suppose we split between $\mathbf{A}_k$ and $\mathbf{A}_{k+1}$, then the parenthesization of $\mathbf{A}_1 \dots \mathbf{A}_k$ and $\mathbf{A}_{k+1} \dots \mathbf{A}_n$ have to also be optimal
- Let $c[1,k]$ and $c[k+1,n]$ be the optimal costs for the subproblems, then the cost of splitting at $k, k+1$ is

$$c[1,k] + c[k+1,n] + p_0 p_k p_n$$

- Thus, the main recurrence is

$$c[1,n] = \min_{1 \le k < n} \left(c[1,k] + c[k+1,n] + p_0 p_k p_n \right)$$

- Hence, in general we need $c[i,j]$ for $i < j$:

$$c[i,j] = \min_{i \le k < j} \left(c[i,k] + c[k+1,j] + p_{i-1} p_k p_j \right)$$

The Recurrence

$$c[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min_{i \le k < j} \left(c[i,k] + c[k+1,j] + p_{i-1} p_k p_j \right) & \text{if } i < j \end{cases}$$

Pseudo Code

- Main Question: how do we fill out the table c?

$\text{MCM-ORDER}(p, n)$

$c[i,i] \leftarrow 0$ for $i = 1, \dots, n$
for $l = 1$ **to** $n - 1$ **do**
  for $i \leftarrow 1$ **to** $n - l$ **do**
    $j \leftarrow i + l$; // not really needed, just to be clearer
    $c[i,j] \leftarrow \infty$;
    for $k \leftarrow i$ **to** $j - 1$ **do**
      $t \leftarrow c[i,k] + c[k+1,j] + p_{i-1}p_kp_j$;
      if $c[i,j] > t$ **then**
        $c[i,j] \leftarrow t$;
      end if
    end for
  end for
end for
return $c[1,n]$;

Constructing the Solution

Use $s[i,j]$ to store the optimal splitting point k:

$\text{MCM-ORDER}(p, n)$

$c[i,i] \leftarrow 0$ for $i = 1, \dots, n$
for $l = 1$ **to** $n - 1$ **do**
  for $i \leftarrow 1$ **to** $n - l$ **do**
    $j \leftarrow i + l$; // not really needed, just to be clearer
    $c[i,j] \leftarrow \infty$;
    for $k \leftarrow i$ **to** $j - 1$ **do**
      $t \leftarrow c[i,k] + c[k+1,j] + p_{i-1}p_kp_j$;
      if $c[i,j] > t$ **then**
        $c[i,j] \leftarrow t$; $s[i,j] \leftarrow k$;
      end if
    end for
  end for
end for
return c, s;

Space and Time Complexity

- Space needed is $O(n^2)$ for the tables c and s
- Suppose the inner-most loop takes about 1 time unit, then the running time is

$$
\begin{aligned}
\sum_{l=1}^{n-1}\sum_{i=1}^{n-l} l &= \sum_{l=1}^{n-1} l(n-l) \\
&= n\sum_{l=1}^{n-1} l - \sum_{l=1}^{n-1} l^2 \\
&= n\frac{n(n-1)}{2} - \frac{(n-1)n(2(n-1)+6)}{6} \\
&= \Theta(n^3)
\end{aligned}
$$

Outline

Knapsack & Subset Sum: Problem Definitions

Knapsack & Subset Sum: Problem Definitions

- SUBSET SUM: given n positive integers $w_1, \dots, w_n$, and a bound W, return a subset of integers whose sum is as large as possible but not more than W

Knapsack & Subset Sum: Problem Definitions

- SUBSET SUM: given n positive integers $w_1, \dots, w_n$, and a bound W, return a subset of integers whose sum is as large as possible but not more than W
- 01-KNAPSACK: given n items with weights $w_1, \dots, w_n$ and corresponding values $v_1, \dots, v_n$, and abound W, find a subset of items with maximum total value whose total weight is bounded by W

Knapsack & Subset Sum: Problem Definitions

- SUBSET SUM: given n positive integers $w_1, \dots, w_n$, and a bound W, return a subset of integers whose sum is as large as possible but not more than W
- 01-KNAPSACK: given n items with weights $w_1, \dots, w_n$ and corresponding values $v_1, \dots, v_n$, and abound W, find a subset of items with maximum total value whose total weight is bounded by W
- SUBSET SUM is a special case of 01-KNAPSACK when $v_i = w_i$ for all i. Thus, we will try to solve 01-KNAPSACK only.

Structure of an Optimal Solution

Structure of an Optimal Solution

- Let $\mathcal{O}$ be an optimal solution, then either the nth item I_n is in $\mathcal{O}$ or not

Structure of an Optimal Solution

- Let $\mathcal{O}$ be an optimal solution, then either the nth item I_n is in $\mathcal{O}$ or not
- If $I_n \in \mathcal{O}$, then $\mathcal{O}' = \mathcal{O} - \{I_n\}$ must be optimal for the problem $\{I_1, \dots, I_{n-1}\}$ with weight bound $W - w_n$

Structure of an Optimal Solution

- Let $\mathcal{O}$ be an optimal solution, then either the nth item I_n is in $\mathcal{O}$ or not
- If $I_n \in \mathcal{O}$, then $\mathcal{O}' = \mathcal{O} - \{I_n\}$ must be optimal for the problem $\{I_1, \dots, I_{n-1}\}$ with weight bound $W - w_n$
- If $I_n \notin \mathcal{O}$, then $\mathcal{O}' = \mathcal{O}$ must be optimal for the problem $\{I_1, \dots, I_{n-1}\}$ with weight bound W

Structure of an Optimal Solution

- Let $\mathcal{O}$ be an optimal solution, then either the nth item I_n is in $\mathcal{O}$ or not
- If $I_n \in \mathcal{O}$, then $\mathcal{O}' = \mathcal{O} - \{I_n\}$ must be optimal for the problem $\{I_1, \dots, I_{n-1}\}$ with weight bound $W - w_n$
- If $I_n \notin \mathcal{O}$, then $\mathcal{O}' = \mathcal{O}$ must be optimal for the problem $\{I_1, \dots, I_{n-1}\}$ with weight bound W
- The above analysis suggests defining $\text{OPT}(j, w)$ to be the optimal value for the problem $\{I_1, \dots, I_j\}$ with weight bound w

The Recurrence and Analysis

$$\text{OPT}(j, w) = \begin{cases} 0 & j = 0 \\ \text{OPT}(j-1, w) & w < w_j \\ \max\{\text{OPT}(j-1, w), v_j + \text{OPT}(j-1, w - w_j)\} & w \geq w_j \end{cases}$$

The Recurrence and Analysis

$$\text{OPT}(j, w) = \begin{cases} 0 & j = 0 \\ \text{OPT}(j-1, w) & w < w_j \\ \max\{\text{OPT}(j-1, w), v_j + \text{OPT}(j-1, w - w_j)\} & w \geq w_j \end{cases}$$

- Running time is $\Theta(nW)$: not polynomial
- This is called pseudo-polynomial time
- 01-Knapsack is **NP**-hard $\Rightarrow$ extremely unlikely to have polynomial-time solution
- However, there exists a poly-time algorithm that returns a feasible solution with value within ϵ of optimality

Outline

Sequence Alignment: Motivation 1

How similar are “ocurrance” and “occurrence”?

Sequence Alignment: Motivation 1

How similar are “ocurrance” and “occurrence”?

o	c	u	r	r	a	n	c	e	-
o	c	c	u	r	r	e	n	c	e

Sequence Alignment: Motivation 1

How similar are "ocurrance" and "occurrence"?

o	c	u	r	r	a	n	c	e	-
o	c	c	u	r	r	e	n	c	e

o	c	-	u	r	r	a	n	c	e
o	c	c	u	r	r	e	n	c	e

Sequence Alignment: Motivation 1

How similar are "ocurrance" and "occurrence"?

o	c	u	r	r	a	n	c	e	-
o	c	c	u	r	r	e	n	c	e

o	c	-	u	r	r	a	n	c	e
o	c	c	u	r	r	e	n	c	e

o	c	-	u	r	r	a	-	n	c	e
o	c	c	u	r	r	-	e	n	c	e

Sequence Alignment: Motivation 2

- Applications in Unix diff program, speech recognition, computational biology
- Edit distance (Levenshtein 1966, Needleman-Wunsch 1970)
  - Gap penalty δ, mismatch penalty α_{pq}
  - Distance or cost equals sum of penalties

A	C	-	A	G	T	A	-	T	G	C
A	C	C	A	T	T	G	T	T	G	C

$$\text{cost} = 2\delta + \alpha_{GT} + \alpha_{AG}$$

Sequence Alignment: Problem Definition

- Given two strings $x_1, \dots, x_m$ and $y_1, \dots, y_n$, find an alignment of minimum cost
- An alignment is a set M of ordered pairs (x_i, y_j) such that each item is in at most one pair and there is no crossing
- Two pairs (x_i, y_j) and (x_p, y_q) cross if $i < p$ but $j > q$

$$\begin{aligned}\text{cost}(M) &= \sum_{(x_i,y_j)\in M} \alpha_{x_i y_j} + \sum_{\text{unmatched } x_i} \delta + \sum_{\text{unmatched } y_i} \delta \\ &= \sum_{(x_i,y_j)\in M} \alpha_{x_i y_j} + \delta(\#\text{unmatched } x_i + \#\text{unmatched } y_j)\end{aligned}$$

Structure of Optimal Solution and Recurrence

Structure of Optimal Solution and Recurrence

- Key observation: either $(x_m, y_n) \in M$ or x_m unmatched or y_n unmatched

Structure of Optimal Solution and Recurrence

- Key observation: either $(x_m, y_n) \in M$ or x_m unmatched or y_n unmatched
- Let $\text{OPT}(i, j)$ be the optimal cost of aligning $x_1, \dots, x_i$ with $y_1, \dots, y_j$, then

Structure of Optimal Solution and Recurrence

- Key observation: either $(x_m, y_n) \in M$ or x_m unmatched or y_n unmatched
- Let $\text{OPT}(i, j)$ be the optimal cost of aligning $x_1, \dots, x_i$ with $y_1, \dots, y_j$, then

$$
\begin{aligned}
\text{OPT}(i, 0) &= i\delta \\
\text{OPT}(0, j) &= j\delta \\
\text{OPT}(i, j) &= \min\{\alpha_{x_i y_j} + \text{OPT}(i-1, j-1), \\
&\qquad \delta + \text{OPT}(i-1, j), \delta + \text{OPT}(i, j-1)\}
\end{aligned}
$$

Time and Space Complexity

- $\Theta(mn)$ for time and space

Time and Space Complexity

- $\Theta(mn)$ for time and space
- Question: for RNA sequences ($m, n \approx 10,000$), $\Theta(mn)$-space is too large, can we do better?

Time and Space Complexity

- $\Theta(mn)$ for time and space
- Question: for RNA sequences ($m, n \approx 10,000$), $\Theta(mn)$-space is too large, can we do better?
- Answer is YES - $\Theta(m+n)$ is possible, due to a beautiful idea by Herschberg in 1975

Time and Space Complexity

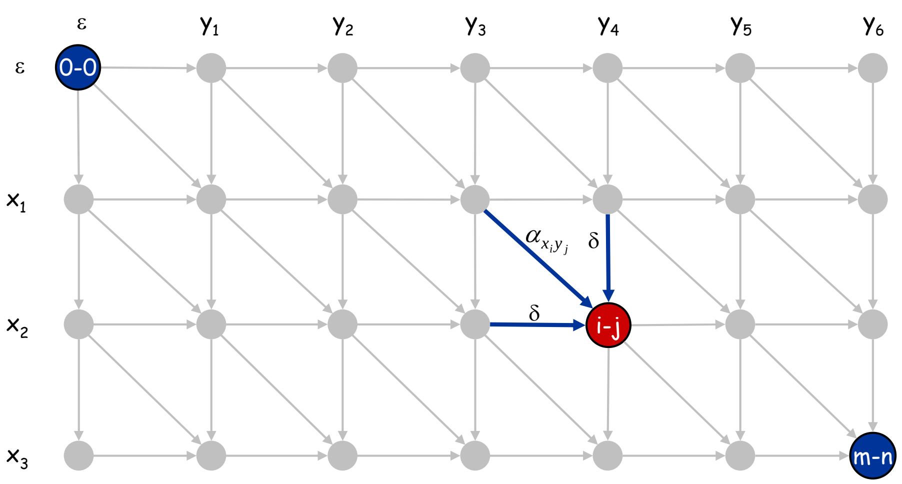
- $\Theta(mn)$ for time and space
- Question: for RNA sequences ($m, n \approx 10,000$), $\Theta(mn)$-space is too large, can we do better?
- Answer is YES - $\Theta(m+n)$ is possible, due to a beautiful idea by Herschberg in 1975
- First attempt: computing $\text{OPT}(m, n)$ using $\Theta(m+n)$-space. How?

Time and Space Complexity

- $\Theta(mn)$ for time and space
- Question: for RNA sequences ($m, n \approx 10,000$), $\Theta(mn)$-space is too large, can we do better?
- Answer is YES - $\Theta(m+n)$ is possible, due to a beautiful idea by Herschberg in 1975
- First attempt: computing $\text{OPT}(m,n)$ using $\Theta(m+n)$-space. How?
- Unfortunately, no easy way to recover the alignment itself.

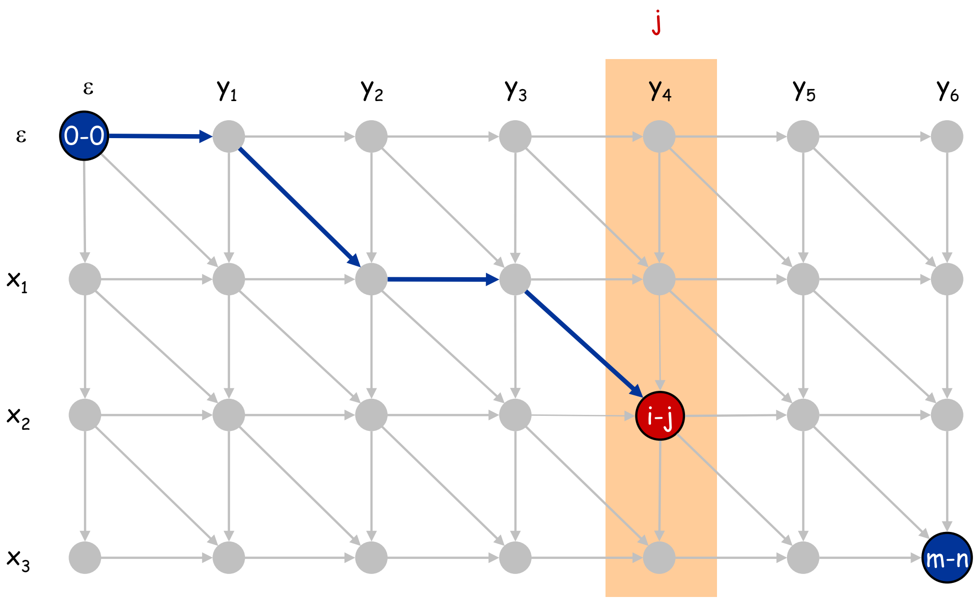
Sequence Alignment in Linear Space

- Herschberg's idea: combine D&C and dynamic programming in a clever way
- Inspired by Savitch's theorem in complexity theory
- Edit Distance Graph: let $f(i,j)$ be the shortest path length from $(0,0)$ to (i,j), then $f(i,j) = \text{OPT}(i,j)$



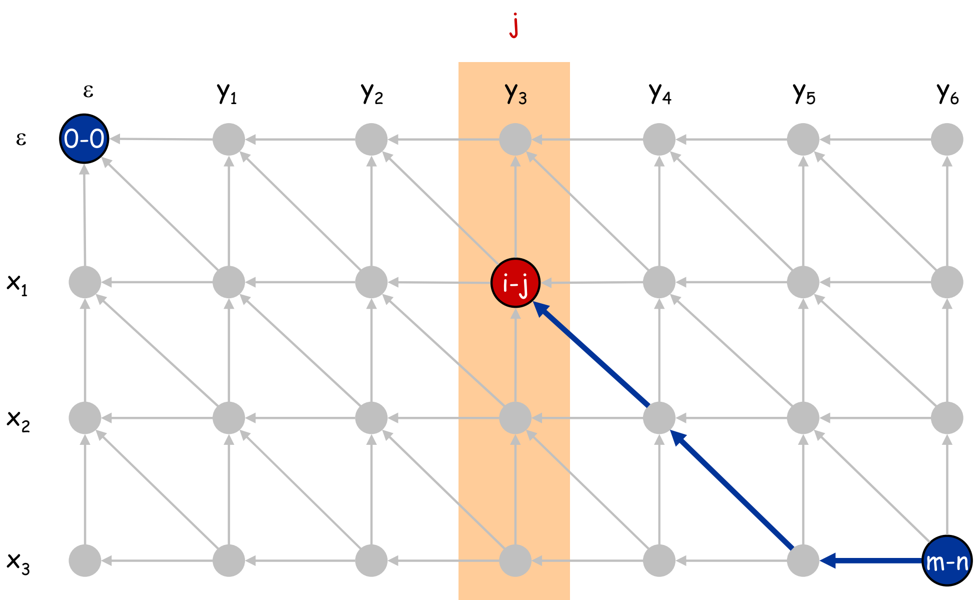
Sequence Alignment in Linear Space

- For any j, can compute $f(\cdot, j)$ in $O(mn)$-time and $O(m+n)$-space



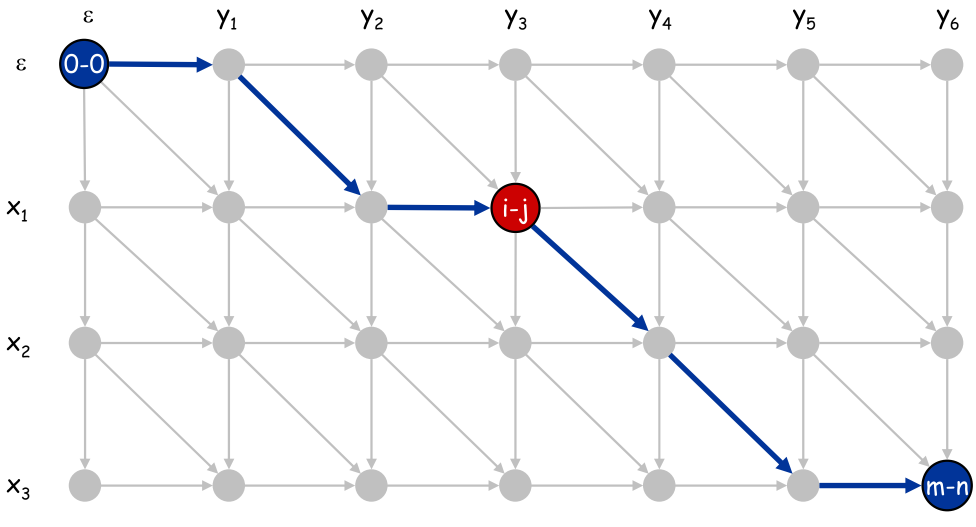
Sequence Alignment in Linear Space

- Let $g(i, j)$ be the shortest distance from (i, j) to (m, n), then $g(\cdot, j)$ can be computed in in $O(mn)$-time and $O(m+n)$-space, for any fixed j



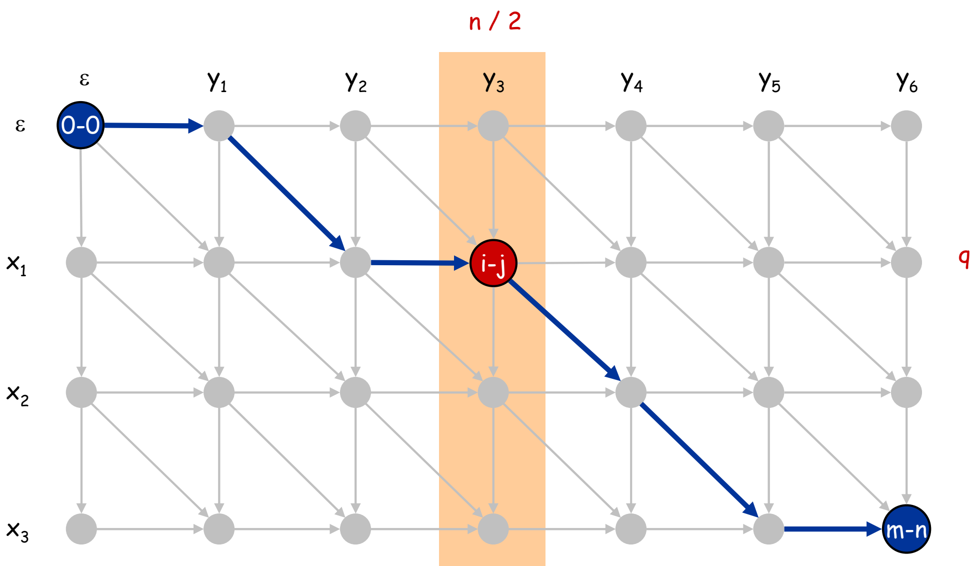
Sequence Alignment in Linear Space

- The cost of a shortest path from $(0,0)$ to (m,n) which goes through (i,j) is $f(i,j)+g(i,j)$



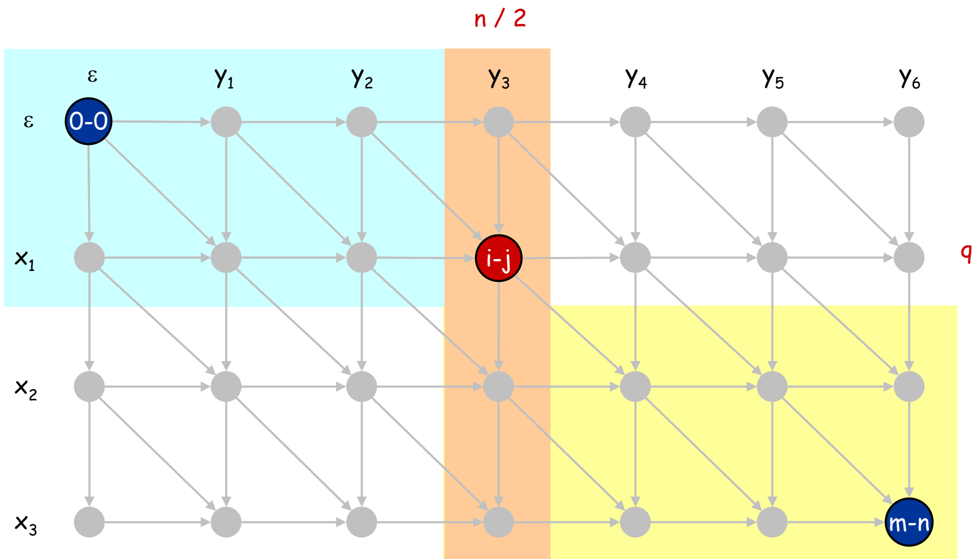
Sequence Alignment in Linear Space

- Let q be an index minimizing $f(q,n/2)+g(q,n/2)$, then a shortest path through $(q,n/2)$ is also a shortest path overall



Sequence Alignment in Linear Space using D&C

- Compute q as described, output $(q, n/2)$, then recursively solve two sub-problems.



Sequence Alignment in Linear Space: Analysis

$$T(m,n) \leq cmn + T(q, n/2) + T(m-q, n/2)$$

Induction gives $T(m,n) = O(mn)$
Thus, the running time remains $O(mn)$, yet space requirement is only $O(m+n)$

Outline

Outline

1. What is Dynamic Programming?
2. Weighted Inverval Scheduling
3. Longest Common Subsequence
4. Segmented Least Squares
5. Matrix-Chain Multiplication (MCM)
6. 01-Knapsack and Subset Sum
7. Sequence Alignment
8. Shortest Paths in Graphs
   - Bellman-Ford Algorithm
   - All-Pairs Shortest Paths

Shortest Path: Problem Definition

- Shortest Path Problem: given a directed graph $G = (V, E)$ with edge cost $c : E \to \mathbb{R}$, find a shortest path from a given source s to a destination t
- Dijkstra's algorithm does not work because there might be negative cycles.
- We will also address the problem of finding a negative cycle (if any).

Structure of an Optimal Solution

Structure of an Optimal Solution

- Consider first the case when there's no negative cycle

Structure of an Optimal Solution

- Consider first the case when there's no negative cycle
- Let $P = s, v_1, \dots, v_{k-1}, t$ be a shortest path from s to t, we can assume (why?) that P is a simple path (i.e. no repeated vertex)

Structure of an Optimal Solution

- Consider first the case when there's no negative cycle
- Let $P = s, v_1, \dots, v_{k-1}, t$ be a shortest path from s to t, we can assume (why?) that P is a simple path (i.e. no repeated vertex)
- Attempt 1: let $\text{OPT}(u,t)$ be the length of a shortest path from u to t, clearly
$$\text{OPT}(u,t) = \min\{\text{OPT}(v,t) \mid (u,v) \in E\}$$

Structure of an Optimal Solution

- Consider first the case when there's no negative cycle
- Let $P = s, v_1, \dots, v_{k-1}, t$ be a shortest path from s to t, we can assume (why?) that P is a simple path (i.e. no repeated vertex)
- Attempt 1: let $\text{OPT}(u,t)$ be the length of a shortest path from u to t, clearly
$$\text{OPT}(u,t) = \min\{\text{OPT}(v,t) \mid (u,v) \in E\}$$

- Problem is, it's not clear how the $\text{OPT}(v,t)$ are "smaller" problems than the original $\text{OPT}(u,t)$. Thus, we need a way to clearly say some $\text{OPT}(v,t)$ are "smaller" than another $\text{OPT}(u,t)$

Structure of an Optimal Solution

- Consider first the case when there's no negative cycle
- Let $P = s, v_1, \dots, v_{k-1}, t$ be a shortest path from s to t, we can assume (why?) that P is a simple path (i.e. no repeated vertex)
- Attempt 1: let $\text{OPT}(u, t)$ be the length of a shortest path from u to t, clearly
$$\text{OPT}(u, t) = \min\{\text{OPT}(v, t) \mid (u, v) \in E\}$$
- Problem is, it's not clear how the $\text{OPT}(v, t)$ are "smaller" problems than the original $\text{OPT}(u, t)$. Thus, we need a way to clearly say some $\text{OPT}(v, t)$ are "smaller" than another $\text{OPT}(u, t)$
- Bellman-Ford: fix target t, let $\text{OPT}(i, u)$ be the length of a shortest path from u to t with at most i edges
- What we want is $\text{OPT}(n - 1, s)$

The Recurrence and Analysis

$$\text{OPT}(i, u) = \begin{cases} 0 & i = 0, u = t \\ \infty & i = 0, u \neq t \\ \min\left\{\text{OPT}(i-1, u), \min\limits_{v:(u,v)\in E}\{\text{OPT}(i-1, v) + c_{uv}\}\right\} & i > 0 \end{cases}$$

- Space complexity is $O(n^2)$
- Time complexity is $O(n^3)$: filling out the $n \times n$ table row by row, top to bottom, computing each entry takes $O(n)$
- Better time analysis: computing $\text{OPT}(i, u)$ takes time $O(\text{out-deg}(u))$, for a total of
$$O\left(n \sum_u \text{out-deg}(u)\right) = O(mn)$$

More Space-Efficient Implementation

More Space-Efficient Implementation

- First Attempt: use a two column table, since $\text{OPT}(i, u)$ only depends on $\text{OPT}(i-1, *)$; thus need $O(n)$-space.

More Space-Efficient Implementation

- First Attempt: use a two column table, since $\text{OPT}(i, u)$ only depends on $\text{OPT}(i-1, *)$; thus need $O(n)$-space.
- Second Attempt: use a one column table. Instead of $\text{OPT}(i, u)$ we only have $\text{OPT}(u)$, using i as the iteration number

Space Efficient Bellman-Ford(G, t)

```
OPT(u) ← ∞, ∀u;   OPT(t) ← 0
for i = 1 to n − 1 do
    for each vertex u do
        OPT(u) ← min { OPT(u), min_{v:(u,v)∈E} {OPT(v) + c_uv} }
    end for
end for
```

$$\text{OPT}(u) \leftarrow \min\left\{\text{OPT}(u), \min_{v:(u,v)\in E}\{\text{OPT}(v) + c_{uv}\}\right\}$$

Why Does Space Efficient Bellman-Ford Work?

- What might be the problem?

Why Does Space Efficient Bellman-Ford Work?

- What might be the problem?
  - Before, $\textsc{opt}(i, u)$ = length of shortest u, t-path with $\leq i$ edges
  - Now, after iteration i, $\textsc{opt}(u)$ may no longer be the length of shortest u, t-path with $\leq i$ edges

Why Does Space Efficient Bellman-Ford Work?

- What might be the problem?
  - Before, $\textsc{opt}(i, u)$ = length of shortest u, t-path with $\leq i$ edges
  - Now, after iteration i, $\textsc{opt}(u)$ may no longer be the length of shortest u, t-path with $\leq i$ edges

- However, by induction we can show

Why Does Space Efficient Bellman-Ford Work?

- What might be the problem?
  - Before, $\text{OPT}(i, u)$ = length of shortest u, t-path with $\leq i$ edges
  - Now, after iteration i, $\text{OPT}(u)$ may no longer be the length of shortest u, t-path with $\leq i$ edges
- However, by induction we can show
  - For any i, if $\text{OPT}(u) < \infty$ then there is a u, t-path with length $\text{OPT}(u)$

Why Does Space Efficient Bellman-Ford Work?

- What might be the problem?
  - Before, $\text{OPT}(i, u)$ = length of shortest u, t-path with $\leq i$ edges
  - Now, after iteration i, $\text{OPT}(u)$ may no longer be the length of shortest u, t-path with $\leq i$ edges
- However, by induction we can show
  - For any i, if $\text{OPT}(u) < \infty$ then there is a u, t-path with length $\text{OPT}(u)$
  - After i iterations, $\text{OPT}(u) \leq \text{OPT}(i, u)$

Why Does Space Efficient Bellman-Ford Work?

- What might be the problem?
  - Before, $\text{OPT}(i, u)$ = length of shortest u, t-path with $\leq i$ edges
  - Now, after iteration i, $\text{OPT}(u)$ may no longer be the length of shortest u, t-path with $\leq i$ edges
- However, by induction we can show
  - For any i, if $\text{OPT}(u) < \infty$ then there is a u, t-path with length $\text{OPT}(u)$
  - After i iterations, $\text{OPT}(u) \leq \text{OPT}(i, u)$
- Consequently, after $n-1$ iterations we have $\text{OPT}(u) \leq \text{OPT}(n-1, u)$, done!

Construction of Shortest Paths

Similar to Dijkstra's algorithm, maintain a pointer $\text{SUCCESSOR}(u)$ for each u, pointing to the next vertex along the current path to t (thus, total space complexity $= O(m+n)$)

$\text{SPACE EFFICIENT BELLMAN-FORD}(G, t)$

```
OPT(u) ← ∞, ∀u;   OPT(t) ← 0
SUCCESSOR(u) ← NIL, ∀u
for i = 1 to n − 1 do
    for each vertex u do
        w ← argmin_{v:(u,v)∈E} {OPT(v) + c_uv}
        if OPT(u) > OPT(w) + c_uw then
            OPT(u) ← OPT(w) + c_uw
            SUCCESSOR(u) ← w
        end if
    end for
end for
```

Detecting Negative Cycles

Detecting Negative Cycles

Lemma

If $\text{OPT}(n, u) = \text{OPT}(n-1, u)$ *for all nodes* u*, then there is no negative cycle on any path from* u *to* t

©Hung Q. Ngo (SUNY at Buffalo) CSE 531 Algorithm Analysis and Design 61 / 74

Detecting Negative Cycles

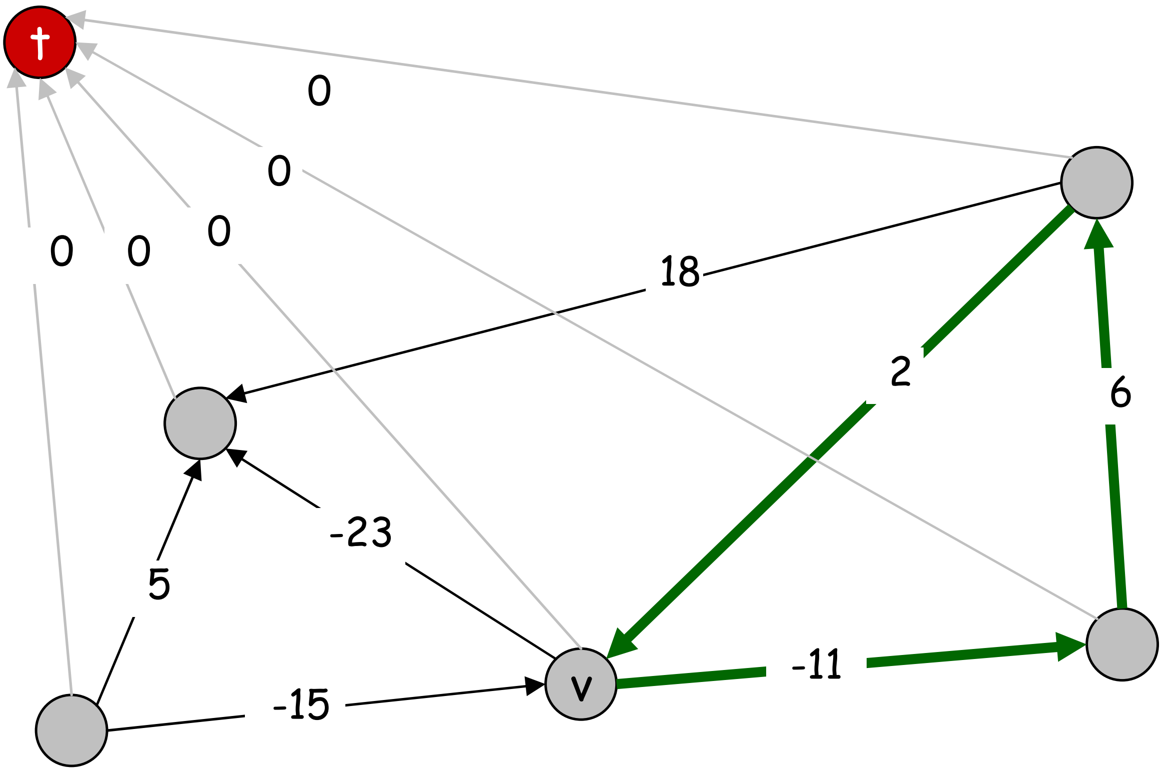
Lemma

If $\text{OPT}(n, u) = \text{OPT}(n-1, u)$ *for all nodes* u*, then there is no negative cycle on any path from* u *to* t

Theorem

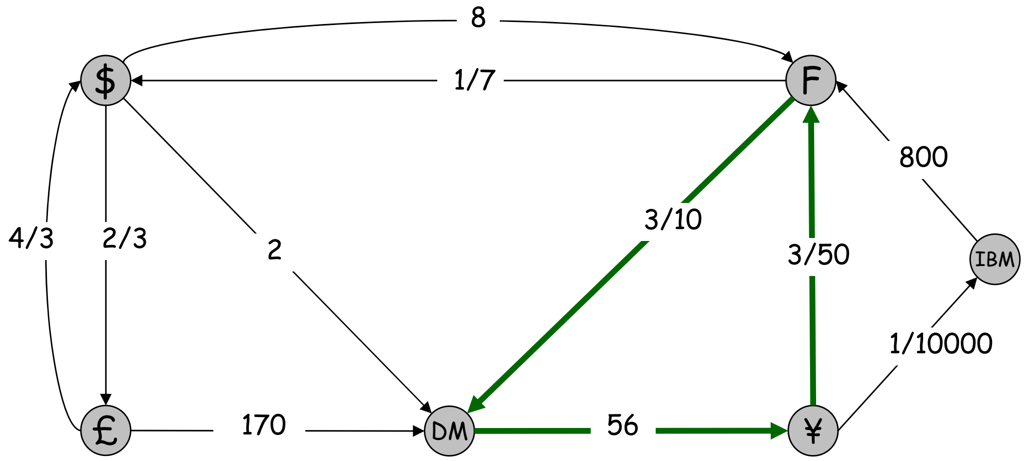
If $\text{OPT}(n, u) < \text{OPT}(n-1, u)$ *for some node* u*, then any shortest path from* u *to* t *contains a negative cycle* C*.*

Detecting Negative Cycles



Detecting Negative Cycles: Application

- Given n currencies and exchange rates between them, is there an arbitrage opportunity?
- Fast algorithm is ... money!



Outline

All-Pairs Shortest Paths: Problem Definition

- Input: directed graph $G = (V, E)$, cost function $c : E \to \mathbb{R}$. Assume no negative cycle.
- Input represented by a cost matrix $\mathbf{C} = (c_{uv})$

$$c_{uv} = \begin{cases} c(uv) & \text{if } uv \in E \\ 0 & \text{if } u = v \\ \infty & \text{otherwise} \end{cases}$$

- Output:
  - a distance matrix $\mathbf{D} = (d_{uv})$, where d_{uv} = shortest path length from u to v, and ∞ otherwise.
  - a predecessor matrix $\mathbf{\Pi} = (\pi_{uv})$, where π_{uv} is v's previous vertex on a shortest path from u to v, and NIL if v is not reachable from u or $u = v$.

A Solution Based on Bellman-Ford's Idea

- $d_{uv}^{(k)}$: length of a shortest path from u to v with $\leq k$ edges ($k \geq 1$)
- Let $\mathbf{D}^{(k)} = (d_{uv}^{(k)})$ (a matrix)
- We can see that $\mathbf{D} = \mathbf{D}^{(n-1)}$, $\mathbf{D}^{(1)} = \mathbf{C}$

A Solution Based on Bellman-Ford's Idea

- $d_{uv}^{(k)}$: length of a shortest path from u to v with $\leq k$ edges ($k \geq 1$)
- Let $\mathbf{D}^{(k)} = (d_{uv}^{(k)})$ (a matrix)
- We can see that $\mathbf{D} = \mathbf{D}^{(n-1)}$, $\mathbf{D}^{(1)} = \mathbf{C}$

Then,

$$\begin{aligned} d_{uv}^{(k)} &= \min_{w \in V, w \neq v} \left\{ d_{uv}^{(k-1)}, d_{uw}^{(k-1)} + c_{wv} \right\} \\ &= \min_{w \in V} \left\{ d_{uw}^{(k-1)} + c_{wv} \right\} \end{aligned}$$

Implementation of the Idea

Use a 3-dimensional table for the $d_{uv}^{(k)}$, how to fill the table?

Implementation of the Idea

Use a 3-dimensional table for the $d_{uv}^{(k)}$, how to fill the table?

Bellman-Ford APSP$(\mathbf{C}, n)$

- $\mathbf{D}^{(1)} \leftarrow \mathbf{C}$ // this actually takes $O(n^2)$
- **for** $k \leftarrow 2$ **to** $n-1$ **do**
  - **for** each $u \in V$ **do**
    - **for** each $v \in V$ **do**
      - $d_{uv}^{(k)} \leftarrow \min_{w \in V}\{d_{uw}^{(k-1)} + c_{wv}\}$
    - **end for**
  - **end for**
- **end for**
- Return $\mathbf{D}^{(n-1)}$ // the last "layer"

Implementation of the Idea

Use a 3-dimensional table for the $d_{uv}^{(k)}$, how to fill the table?

Bellman-Ford APSP$(\mathbf{C}, n)$

- $\mathbf{D}^{(1)} \leftarrow \mathbf{C}$ // this actually takes $O(n^2)$
- **for** $k \leftarrow 2$ **to** $n-1$ **do**
  - **for** each $u \in V$ **do**
    - **for** each $v \in V$ **do**
      - $d_{uv}^{(k)} \leftarrow \min_{w \in V}\{d_{uw}^{(k-1)} + c_{wv}\}$
    - **end for**
  - **end for**
- **end for**
- Return $\mathbf{D}^{(n-1)}$ // the last "layer"

- $O(n^4)$-time, $O(n^3)$-space.
- Space can be reduced to $O(n^2)$, how?

Some Observations

- $\mathbf{\Pi}$ can be updated at each step as usual
- Ignoring the outer loop, replace $\min$ by $\sum$ and $+$ by $\cdot$, the previous code becomes

```
1: for each u ∈ V do
2:     for each v ∈ V do
3:         d_uv^(k) ← ∑_{w∈V} d_uw^(k-1) · c_wv
4:     end for
5: end for
```

- This is like $\mathbf{D}^{(k)} \leftarrow \mathbf{D}^{(k-1)} \odot \mathbf{C}$, where $\odot$ is identical to matrix multiplication, except that $\sum$ replaced by $\min$, and $\cdot$ replaced by $+$
- $\mathbf{D}^{(n-1)}$ is just $\mathbf{C} \odot \mathbf{C} \cdots \odot \mathbf{C}$, $n-1$ times.
- It is easy (?) to show that $\odot$ is associative
- Hence, $\mathbf{D}^{(n-1)}$ can be calculated from $\mathbf{C}$ in $O(\lg n)$ steps by "repeated squaring," for a total running time of $O(n^3 \lg n)$

Floyd-Warshall's Idea

- Write $V = \{1, 2, \dots, n\}$
- Let $d_{ij}^{(k)}$ be the length of a shortest path from i to j, all of whose intermediate vertices are in the set $[k] := \{1, \dots, k\}.0 \leq k \leq n$
- We agree that $[0] = \emptyset$, so that $d_{ij}^{(0)}$ is the length of a shortest path between i and j with no intermediate vertex.
- Then, we get the following recurrence:

$$d_{ij}^{(k)} = \begin{cases} c_{ij} & \text{if } k = 0 \\ \min\left\{\left(d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right), d_{ij}^{(k-1)}\right\} & \text{if } k \geq 1 \end{cases}$$

- The matrix we are looking for is $D = D^{(n)}$.

Pseudo Code for Floyd-Warshall Algorithm

Floyd-Warshall$(\mathbf{C}, n)$

```
D^(0) ← C
for k ← 1 to n do
    for i ← 1 to n do
        for j ← 1 to n do
            d_ij^(k) ← min{(d_ik^(k-1) + d_kj^(k-1)), d_ij^(k-1)}
        end for
    end for
end for
Return D^n // the last "layer"
```

Time: $O(n^3)$, space: $O(n^3)$.

Constructing the Π matrix

$$\pi_{ij}^{(0)} = \begin{cases} \text{NIL} & \text{if } i = j \text{ or } c_{ij} = \infty \\ i & \text{otherwise} \end{cases}$$

and for $k \geq 1$

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{if } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ \pi_{kj}^{(k-1)} & \text{if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{cases}$$

Question: is it correct if we do

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{if } d_{ij}^{(k-1)} < d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ \pi_{kj}^{(k-1)} & \text{if } d_{ij}^{(k-1)} \geq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{cases}$$

Finally, $\mathbf{\Pi} = \mathbf{\Pi}^{(n)}$.

Floyd-Warshall with Less Space

SPACE EFFICIENT FLOYD-WARSHALL($\mathbf{C}, n$)

```
D ← C
for k ← 1 to n do
    for i ← 1 to n do
        for j ← 1 to n do
            d_ij ← min{(d_ik + d_kj), d_ij}
        end for
    end for
end for
Return D
```

Time: $O(n^3)$, space: $O(n^2)$.

Why does this work?

Application: Transitive Closure of a Graph

- Given a directed graph $G = (V, E)$
- We'd like to find out whether there is a path between i and j for every pair i, j.
- $G^* = (V, E^*)$, the transitive closure of G, is defined by

$$ij \in E^* \text{ iff there is a path from } i \text{ to } j \text{ in } G.$$

- Given the adjacency matrix $\mathbf{A}$ of G ($a_{ij} = 1$ if $ij \in E$, and 0 otherwise)
- Compute the adjacency matrix $\mathbf{A}^*$ of $\mathbf{G}^*$

Transitive Closure with Dynamic Programming

- Let $a_{ij}^{(k)}$ be a boolean variable, indicating whether there is a path from i to j all of whose intermediate vertices are in the set $[k]$.
- We want $\mathbf{A}^* = \mathbf{A}^{(n)}$.
- Note that

$$a_{ij}^{(0)} = \begin{cases} \text{TRUE} & \text{if } ij \in E \text{ or } i = j \\ \text{FALSE} & \text{otherwise} \end{cases}$$

and for $k \geq 1$

$$a_{ij}^{(k)} = a_{ij}^{(k-1)} \vee (a_{ik}^{(k-1)} \wedge a_{kj}^{(k-1)})$$

- Time: $O(n^3)$, space $O(n^3)$

Transitive Closure with Dynamic Programming

- Let $a_{ij}^{(k)}$ be a boolean variable, indicating whether there is a path from i to j all of whose intermediate vertices are in the set $[k]$.
- We want $\mathbf{A}^* = \mathbf{A}^{(n)}$.
- Note that

$$a_{ij}^{(0)} = \begin{cases} \text{TRUE} & \text{if } ij \in E \text{ or } i = j \\ \text{FALSE} & \text{otherwise} \end{cases}$$

and for $k \geq 1$

$$a_{ij}^{(k)} = a_{ij}^{(k-1)} \vee (a_{ik}^{(k-1)} \wedge a_{kj}^{(k-1)})$$

- Time: $O(n^3)$, space $O(n^3)$
- So what's the advantage of doing this instead of Floyd-Warshall?